

High-Speed Cryptography and Cryptanalysis

Peter Schwabe

High-Speed Cryptography and Cryptanalysis

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 24 januari 2011 om 16.00 uur

door

Peter Schwabe

geboren te Siegburg, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. T. Lange

en

prof.dr. D.J. Bernstein

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Schwabe, Peter

High-Speed Cryptography and Cryptanalysis /
door Peter Schwabe. –

Eindhoven: Technische Universiteit Eindhoven, 2011

Proefschrift. – ISBN 978-90-386-2415-0

NUR 919 Subject heading: Cryptology

2000 Mathematics Subject Classification: 94A60, 94A62, 11Y16, 11T71

Printed by Printservice Technische Universiteit Eindhoven

Cover design by Paula Schwabe

Public domain

Für meine Eltern

Promotor:

prof.dr. T. Lange

prof.dr. D.J. Bernstein (University of Illinois at Chicago)

Commissie:

prof.dr. A.M. Cohen, chairman

prof.dr. B. Preneel (Katholieke Universiteit Leuven)

prof.dr. M. Scott (Dublin City University)

prof.dr.ir. H.C.A. van Tilborg

prof.dr. G. Woeginger

prof.dr. B.-Y. Yang (Academia Sinica)

Thanks

This thesis would not exist without the support, help and encouragement of many people.

First of all I would like to thank my supervisors Tanja Lange and Daniel J. Bernstein. The time and effort they spent on teaching, guiding and supporting me is probably best expressed by using the German words for Ph.D. supervisor and saying that they are really a “Doktormutter” and “Doktorvater”.

I also want to express my gratitude to Michael Naehrig. He supervised my Diplomarbeit, taught me a lot about elliptic curves and pairings during our joint time in Aachen and Eindhoven, introduced me to my supervisors Tanja Lange and Daniel J. Bernstein, and always supported and encouraged me. I am very grateful for Michael’s friendship.

I thank Arjeh Cohen, Bart Preneel, Michael Scott, Henk van Tilborg, Gerhard Woeginger, and Bo-Yin Yang for joining my Ph.D. committee and for reading this manuscript and giving valuable comments.

Various people invited me to present my results or to work together, I am very grateful for these opportunities and wish to thank Pierrick Gaudry, Emmanuel Thomé, Jérémie Detrey, and Gaëtan Bisson at INRIA, Nancy; Bart Preneel and Emilia Käsper at Katholieke Universiteit Leuven; Christof Paar, Tim Güneysu, and Timo Kasper at Ruhr Universität Bochum; and Francisco Rodríguez-Henríquez, Debrup Chakraborty, Luis Gerardo de la Fraga, and Jorge E. González Díaz at CINEVESTAV, Mexico City. Especially I want to thank Chen-Mou Cheng at National Taiwan University and Bo-Yin Yang at Academia Sinica who invited me several times to Taipei and also hosted me for many weeks while I was writing this thesis. I also want to thank Grace Song for welcoming me as a frequent visitor in the apartment in Taipei that she shares with my girlfriend Chen-Han Lee.

I thank my coauthors Gerd Ascheid, Dominik Auras, Daniel V. Bailey, Brian Baldwin, Lejla Batina, Paulo S. L. M. Barreto, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Neil Costigan, Gauthier van Damme, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gürkaynak, David Kammeler, Emilia Käsper, Thorsten Kleinjung, Tanja Lange, Markus Langenberg, Rainer Leupers, Nele Mentens, Heinrich Meyr, Rudolf Mathar, Giacomo de Meulenaer, Michael Naehrig, Ruben Niederhagen, Christof Paar, Christiane Peters, Francesco Regazzoni, Hanno Scharwaechter, Leif Uhsadel, Anthony Van Herrewege, Bo-Yin Yang, and Dian-dian Zhang for the fruitful collaboration.

I thank the European Commission for supporting my Ph.D studies through the ICT Programme under Contract ICT-2007-216499 CACE.

Many thanks go to Michael Naehrig, Ruben Niederhagen, Christiane Peters and Anne Schwabe for proofreading earlier versions of this thesis, pointing out mistakes and suggesting improvements.

I wish to thank the people at the coding and cryptology group at Eindhoven University of Technology, in particular Henk van Tilborg and Anita Klooster-Derks for making this group a very pleasant work environment. I also want to thank the other EiPSI Ph.D. students, former and current, for their company: Peter Birkner, Gaëtan Bisson, Dion Boesten, Mayla Brusò, Elisa Costante, Sebastiaan de Hoogh, Relinde Jurrius, Mehmet Sabir Kiraz, Xiao-Ping Liang, Peter van Liesdonk, Michael Naehrig, Ruben Niederhagen, Jan-Jaap Osterwijk, Jing Pan, Christiane Peters, Bruno Pontes Soares Rocha, Reza Rezaeian Farashahi, Antonino Simone, Daniel Trivellato, Meilof Veeningen, and José Villegas. I also thank Çiçek Güven, Maxim Hendriks, and Shona Yu for the company in the coffee breaks and for the great time in Istanbul.

Let me also mention the people at the Institute for Theoretical Information Technology at RWTH Aachen University: Rudolf Mathar, Mathilde Getz, Detlef Maus, Daniel Bielefeld, Georg Böcherer, Daniel Catrein, Alexander Engels, Gernot Fabeck, Chunhui Liu, Wolfgang Meyer zu Bergsten, Natalie Naehrig, Melanie Neunerdt, Michael Reyer, Tobias Rick, Anke Schmeink, Michael Schmeink, and Milan Zivkovic. Thank you for the support and the good time I had working together with you in Aachen.

I very much thank my parents and my sister for their support. I thank my cousin Paula Schwabe for designing the cover of this thesis.

Finally I deeply thank my girlfriend Chen-Han Lee for her support, understanding, and for her love.

Contents

List of algorithms	13
List of tables	15
1 Introduction	17
2 Preliminaries	23
2.1 Instructions and instruction sets	24
2.2 Exploiting instruction-level parallelism	25
2.2.1 Dependencies between instructions	25
2.2.2 Pipelined and superscalar execution	26
2.2.3 Optimizing code for in-order CPUs	27
2.2.4 Optimizing code for out-of-order CPUs	28
2.3 Exploiting data-level parallelism	28
2.3.1 Reducing to instruction-level parallelism	29
2.3.2 Single instruction stream, multiple data streams (SIMD)	29
2.3.3 Algorithmic improvements	31
2.4 Exploiting task-level parallelism	31
2.5 Function calls, loops, and conditional statements	32
2.6 Accessing memory	35
2.6.1 Caching	35
2.6.2 Virtual-address translation	37
2.7 Limitations of compilers	38
2.8 The qhasm programming language	38
3 Implementations of AES	41
3.1 The Advanced Encryption Standard (AES)	42
3.1.1 AES encryption	42
3.1.2 Combining SUBBYTES, SHIFTRows, and MIXCOLUMNS	44
3.1.3 AES key expansion	44
3.1.4 The Galois/Counter Mode (GCM)	44
3.2 Cache-timing attacks against AES and GCM	45
3.2.1 Attacks against AES encryption	46
3.2.2 Attacks against AES key expansion	46
3.2.3 Attacks against Galois/Counter Mode authentication	47
3.3 Table-based implementations	48

3.3.1	Reducing instructions for table-based AES	48
3.3.2	Reducing cycles for table-based AES	54
3.4	Bitsliced AES for 64-bit Intel processors	57
3.4.1	Bitsliced representation of the AES state	58
3.4.2	The ADDROUNDKEY operation	58
3.4.3	The SUBBYTES operation	58
3.4.4	The SHIFTRows operation	59
3.4.5	The MIXCOLUMNS operation	60
3.4.6	Key expansion	61
3.5	AES-GCM for 64-bit Intel processors	61
3.5.1	Table-based implementation	61
3.5.2	Constant-time implementation	62
3.6	Performance results and comparison	63
3.6.1	Benchmarks of AES-CTR	65
3.6.2	Benchmarks of AES-GCM	72
4	Elliptic-curve cryptography on Cell processors	75
4.1	ECDH key exchange and the Curve25519 function	76
4.1.1	Elliptic-Curve Diffie-Hellman key exchange (ECDH)	76
4.1.2	Montgomery ladder for scalar multiplication	77
4.1.3	The Curve25519 function	77
4.2	Implementation of Curve25519	78
4.2.1	Multiplication and squaring	80
4.2.2	Reduction	82
4.2.3	Montgomery ladder step	83
4.3	Performance results and comparison	86
5	Pairing computation on AMD64 processors	89
5.1	Background on cryptographic pairings	90
5.2	An optimal ate pairing on Barreto-Naehrig curves	91
5.3	High-level techniques	93
5.3.1	Field extensions	94
5.3.2	Miller loop	94
5.3.3	Final exponentiation	94
5.4	Mid-level techniques: arithmetic in \mathbb{F}_{p^2} and \mathbb{F}_p	95
5.4.1	Representing base field elements	95
5.4.2	Multiplication modulo p	96
5.5	Low-level techniques: using SIMD floating-point arithmetic	97
5.5.1	Avoiding overflows	99
5.5.2	Implementation of field arithmetic	100
5.6	Performance results and comparison	102
5.6.1	Comparison with previous work	102
5.6.2	Comparison with [BGM ⁺ 10]	106

6	Solving ECC2K-130	109
6.1	The parallel version of Pollard's rho algorithm	110
6.2	ECC2K-130 and choice of the iteration function	111
6.2.1	Computing the iteration function	112
6.2.2	Bitsliced binary-field arithmetic	112
6.2.3	Representing elements of $\mathbb{F}_{2^{131}}$	113
6.3	Implementing the iteration function on Cell processors	114
6.4	Implementing the iteration function on NVIDIA GPUs	118
6.4.1	Programming the NVIDIA GTX 295 graphics card	118
6.4.2	131-coefficient binary-polynomial multiplication	120
6.4.3	ECC2K-130 iterations on the GPU	123
6.5	Performance results and comparison	126
7	Implementing Wagner's generalized birthday attack	129
7.1	Wagner's generalized birthday attack	130
7.1.1	Wagner's tree algorithm	130
7.1.2	Wagner in storage-restricted environments	131
7.2	The Fast Syndrome-Based hash function (FSB)	133
7.2.1	Details of the FSB hash function	133
7.2.2	Attacking the compression function of FSB_{48}	134
7.3	Attack strategy	135
7.3.1	How large is a list entry?	135
7.3.2	What list size can be handled with 5.5 TB of storage?	136
7.3.3	The strategy	137
7.4	Implementing the Attack	139
7.4.1	Parallelization	139
7.4.2	Efficient implementation	140
7.5	Results	143
7.5.1	Cost estimates	143
7.5.2	Cost measurements	143
7.5.3	Time-storage tradeoffs	144
7.6	Scalability Analysis	144
	Bibliography	147

List of algorithms

1	AES-128 encryption	43
2	AES-128 key expansion	45
3	AES-GCM encryption and authentication	45
4	Multiplication in $\mathbb{F}_{2^{128}}$ of D with a constant element H	62
5	The Montgomery ladder for x -coordinate-based scalar multiplication on the elliptic curve $E : By^2 = x^3 + Ax^2 + x$	77
6	One ladder step of the Montgomery ladder	78
7	Structure of the modular reduction in $\mathbb{F}_{2^{255}-19}$	83
8	Structure of a Montgomery ladder step (see Algorithm 6) optimized for 4-way parallel computation	85
9	Optimal ate pairing on BN curves for $u > 0$	92
10	Exponentiation by $v = 1868033$	95
11	Degree reduction after polynomial multiplication	97
12	Coefficient reduction	98

List of tables

3.1	Instruction count for the AES S-Box	59
3.2	Machines used for benchmarking the implementations of AES and AES-GCM	63
3.3	Cycles/byte for AES-CTR encryption on gggg and literature claims on PowerPC G4 processors	65
3.4	Cycles/byte for AES-CTR encryption on fireball and literature claims on Intel Pentium 4 processors	66
3.5	Cycles/byte for AES-CTR encryption on smirk and literature claims on UltraSparc II processors	67
3.6	Cycles/byte for AES-CTR encryption on nmi-0039	67
3.7	Cycles/byte for AES-CTR encryption on latour and literature claims on 65-nm Intel Core 2 processors	68
3.8	Cycles/byte for AES-CTR encryption on berlekamp	69
3.9	Cycles/byte for AES-CTR encryption on dragon3	70
3.10	Cycles/byte for AES-CTR encryption on mace and literature claims on 64-bit AMD processors	71
3.11	Cycles/byte for AES-GCM on latour	72
3.12	Cycles/byte for AES-GCM on berlekamp	73
3.13	Cycles/byte for AES-GCM on dragon3	73
4.1	Machines used for benchmarking the Curve25519 implementation for Cell processors	87
4.2	Cycle counts of the Curve25519 software on different machines	87
5.1	Machines used for benchmarking the pairing implementation	103
5.2	Cycle counts of various operations involved in the pairing computation on latour	104
5.3	Cycle counts of various operations involved in the pairing computation on berlekamp	104
5.4	Cycle counts of various operations involved in the pairing computation on dragon3	105
5.5	Cycle counts of various operations involved in the pairing computation on chukonu	105
5.6	Cycle counts (median) of the implementation presented in [BGM ⁺ 10]	106

6.1	Cycle counts and bit-operations for the building blocks used by the Cell and the GPU implementations of the ECC2K-130 iteration function	128
7.1	Parameters of the FSB variants and estimates for the cost of generalized birthday attacks against the compression function.	145

1

Introduction

Cryptology, the “science of the secret”, from Greek κρυπτός (secret) and λόγος (word, science) traditionally encompasses two highly related areas, namely *cryptography* (Greek γράφειν: to write) and *cryptanalysis*. Cryptography deals with encrypting messages, i.e., transforming message plaintexts into ciphertexts under the use of a key in such a way that the plaintext can be retrieved from the ciphertext only with knowledge of the key. Cryptanalysis is the counterpart to cryptography and deals with retrieving plaintexts from given ciphertexts without knowledge of the key.

When the predecessors of modern computers were developed in the first half of the 20th century, cryptography and cryptanalysis were among the first applications of automated computing. Examples of early specialist “computers” for cryptography are the Enigma encryption machines invented by Scherbius and patented in 1928 [Sch28]. The most famous model of the Enigma, the Wehrmacht Enigma, was used by the German troops during World War II. An example for early specialist “computers” for cryptanalysis are the so-called Bombes invented by Turing that were used in the United Kingdom’s main decryption facility in Bletchley Park to break the ciphertexts generated by the Wehrmacht Enigma. Also the first electronic digital information processing machine, the Colossus, was built for cryptanalysis and used in Bletchley Park starting in 1943.

Since these days both fields, automated computing and cryptology, have evolved dramatically. The invention of the von Neumann architecture [vN45, vN93] in 1945 and of the transistor in 1947 [BB50, Sho51, Nob], the first personal computer in 1981, and the long-term trend of exponentially increasing number of components on integrated circuits [Moo65] are only some of the important achievements that led to the computers used today.

The most ground-breaking change in cryptography in that period was certainly

the invention of public-key cryptography by Diffie and Hellman in 1976 [DH76]. Within only a few years research in cryptography picked up topics such as, for example, key-exchange protocols, asymmetric encryption, digital signatures, cryptographic hash functions, and message authentication. This broad variety of primitives enabled the construction of high-level cryptographic protocols, for example secure multi-party computation or zero-knowledge proofs.

Development of computer technology and advances in cryptology were not independent. Not only is a lot of research devoted to special-purpose hardware for cryptographic and cryptanalytical applications, cryptographic applications also influenced and still influence the design of general-purpose computers. The most recent example is Intel's decision to support the Advanced Encryption Standard [NIS01] through special hardware on various Core i7 and Core i5 processors [Gue08].

Even more is research in cryptology driven by advances in computer architecture. The performance of known attacks on general-purpose computers is one of the most important parameters to evaluate the security of cryptographic schemes and choose key sizes appropriately [ECR09, BBB⁺07]. Furthermore, in the design and standardization process, software speed of new primitives has become one of the key properties to decide for or against certain proposals. One reason why the Rijndael cipher was chosen as the Advanced Encryption Standard [NIS01] was because "Rijndael provides consistently high-end performance for encryption, decryption and key setup" [NBB⁺00]. Also for the standardization of the SHA-3 cryptographic hash algorithm, performance in software on standard 32-bit and 64-bit processors is, beside security, one of the most important criteria [NIS07, Section 4].

Why is software performance so important for cryptography? The reason is that many applications require fast cryptographic software and that even small speedups justify high effort. Consider for example Internet content providers running large server farms. Encrypting all transmitted data requires many computers that do nothing but perform cryptographic operations. Even a speedup of only 10% of the software saves 10% of hardware and power cost. Also private users benefit from fast cryptographic software. Consider, for example, the use of an encrypted hard disk in a laptop. Certainly, data transfer to and from the hard disk should not be bottlenecked by cryptography; encryption throughput has to be at least as high as hard-disk throughput. But more than that, more efficient cryptographic implementations leave more processor resources to other programs and help save battery by leaving the processor idle more often.

There are many more examples for the importance of fast cryptographic software and consequently a lot of research is devoted to making cryptographic and cryptanalytical primitives and protocols run fast on standard computers. This area of research is the topic of this thesis.

High-speed cryptography

The term high-speed cryptography as used in this thesis refers to the design and implementation of secure and fast cryptographic software for off-the-shelf computers.

It does *not* refer to the design of new (fast) cryptographic primitives; it does *not* refer to the design of special hardware for cryptography.

Designing and implementing such secure and fast cryptographic software requires careful choices of high-level cryptographic parameters, low-level optimization of software on the assembly level for a given microarchitecture, and considerations of the subtle interactions between high-level and low-level optimizations.

For asymmetric cryptography, high level parameters include for example suitable elliptic curves for elliptic-curve cryptography and pairing-based cryptography, representation of points on these curves, and algorithms for exponentiation. Usually for symmetric cryptographic primitives there are fewer choices to make on the high level, but for example the choice of a mode of operation for block ciphers not only influences security properties but also can have a significant impact on performance.

Optimizations on assembly level include the decision about what type of registers (for example integer registers, vector registers or floating-point registers) are used to implement finite-field arithmetic, the choice of appropriate machine instructions to implement higher-level algorithms, and generic assembly optimizations such as instruction scheduling and register allocation.

These levels of optimizations are not at all independent. One example of interaction between these levels is the choice of an elliptic curve over a certain finite field for elliptic-curve cryptography. High-speed software takes into account specifics of the target computer architecture that allow for particularly efficient implementation of arithmetic in this finite field. Another example is the choice of reduction polynomials for binary finite fields: Scott showed in [Sco07] that trinomials, a choice that seems very obviously optimal when only considering high-level algorithmic aspects, are in fact not optimal for most real-world computers. There are many such interactions between different levels of optimization, some influencing not only performance but also security of implementations.

High-speed cryptanalysis

Analogous to the definition of the term high-speed cryptography, the term high-speed cryptanalysis as used in this thesis refers to the design and implementation of fast cryptanalytical software for off-the-shelf computers to break (or analyze the security of) specific instances of cryptographic primitives. The term thus does *not* refer to finding previously unknown vulnerabilities in cryptographic systems or developing new cryptanalytical methods; it does *not* refer to developing hardware to attack cryptographic systems.

Optimizing cryptographic software and optimizing cryptanalytical software have many things in common, but there are also differences. High-level choices usually do not involve the choice of underlying mathematical structures; these are given by the target cryptosystem. Examples of high-level decisions for cryptanalytical software are the choice of the iteration function in Pollard's rho algorithm, or data-compression techniques for algorithms that involve large amounts of data. Furthermore, cryptanalytical applications require much more computational effort than cryptographic

applications. Most attacks therefore need efficient parallelization to become feasible. In some cases, parallelizing algorithms is straight forward but in many cases more effort is required to exploit the computational power of multi-core systems and computing clusters. Another difference from cryptographic software is that cryptanalytical software does not have to care about security; this makes some optimizations possible that have to be avoided in cryptographic software. More specifically, cryptanalytical implementations do not have to be secured against so-called side-channel attacks.

Overview

Chapter 2 gives the general background on computer architecture and assembly programming required in the remainder of the thesis. It introduces different levels of parallelism in computer programs and describes techniques to exploit these to make programs run faster. Furthermore this chapter explains the effects of function calls, loops, conditional statements in programs, and memory access. Chapter 2 also describes the most important aspects of the qasm programming language which is used for most high-speed software described in this thesis. Details of architectures and microarchitectures are introduced in each of the following chapters as needed to make the chapters self-contained, although this requires repeating some information when implementations described in different chapters target the same architecture.

Chapter 3 describes different implementations of the Advanced Encryption Standard (AES) and of AES-GCM, a mode of operation for combined encryption and authentication [MV04]. This chapter is based on joint work with Bernstein published in [BS08] and with Käsper published in [KS09].

Chapter 4 describes an implementation of elliptic-curve Diffie-Hellman key exchange for the Synergistic Processor Units of the Cell Broadband Engine. This chapter is based on joint work with Costigan published in [CS09].

Chapter 5 describes an implementation of the optimal ate pairing over a Barreto-Naehrig curve targeting Intel Core 2 processors and also running at high speed on other AMD64 CPUs. This chapter is based on joint work with Naehrig and Niederhagen published in [NNS10].

Chapter 6 describes implementations of the parallel version of Pollard's rho algorithm to solve the elliptic-curve discrete-logarithm problem specified by Certicom's challenge ECC2K-130 [Cer97a, Cer97b]. One implementation of the iteration function targets the Synergistic Processor Units of the Cell Broadband Engine, this is joint work with Bos, Kleinjung, and Niederhagen published in [BKNS10]. The other implementation targets NVIDIA GPUs, in particular the NVIDIA GTX 295 graphics card, this is joint work with Bernstein, Chen, Cheng, Lange, Niederhagen and Yang published in [BCC⁺10].

Chapter 7 describes a parallel implementation of Wagner's generalized birthday attack [Wag02a, Wag02b] against the compression function of the toy version FSB₄₈ of the SHA-3 round-1 candidate hash function FSB [AFG⁺09]. This chapter is based on joint work with Bernstein, Lange, Niederhagen, and Peters published in [BLN⁺09].

All software described in this thesis is in the public domain. It is available for download at <http://cryptojedi.org/users/peter/thesis/>.

2

Preliminaries

Most software today is implemented in high-level programming languages, including compiled languages such as C and C++, interpreted languages such as Perl and PHP, and languages that use just-in-time compilation or interpretation of byte code such as Java or Python. All these languages have in common that the source code as the programmer sees it is very different from the program as the computer sees it. This abstraction of programming languages from the actual hardware allows programmers to develop software faster, and write software which is easy to maintain and easily portable to run on different computers.

The disadvantage of implementing software in high-level languages is performance. Even after 60 years of research on compilers, programs compiled from high-level source code usually run significantly slower on a computer than an optimal program could run. How large this loss in performance is depends on the choice of the high-level programming language, the implemented algorithms, what compilers or interpreters are used, and on the abilities and will of a programmer to optimize the program in the given programming language. Section 2.7 will explain why compilers fall short on generating optimal program code.

For most software some performance loss is not a big problem and is outweighed by the benefits of high-level programming languages described above. For the software described in this thesis execution speed is the critical aspect and systematic performance penalties incurred by the use of more convenient high-level programming languages are not acceptable.

The alternative to using high-level languages is to directly implement a program as the computer sees it; this usually means using *assembly language* which can be seen as a human-readable form of the *machine-language* program that runs on the computer.

Optimizing software in assembly requires an understanding of how the computer is going to execute the program. This chapter gives background on computer architecture with a focus on understanding how processors execute programs and how they interact with memory. Furthermore it introduces notation related to computer architecture used in this thesis.

2.1 Instructions and instruction sets

A program as seen by the computer is nothing but a sequence of *instructions*. An instruction is a small building block, usually as simple as “pick up the value at location a , pick up the value at location b , add these two values, and write the result to location c ”. The locations an instruction receives as arguments can be either addresses in memory or *registers*. Registers are small, very fast storage units which belong to the processor. Their size fits the size of data which can be modified in one instruction. For example, a 32-bit integer addition instruction operates on 32-bit registers.

An important difference between registers and main memory, aside from size and speed, is that registers can typically only be *statically addressed*. That means that it is possible to have an instruction such as “add the values in register 1 and register 2 and write the result to register 3”; it is not possible to do something like “look up the value i in register 1 and then do something involving register i ”. All register addresses have to be independent of any input; they are fixed when the program is written. In other words, registers are accessed through *register names* instead of register addresses. Other ways of addressing registers exist—examples are the ICT 1900 computers and the x86 floating-point stack—but they are not relevant for the software described in this thesis.

Which instructions a computer can execute is determined by the *instruction set*; the register names a computer supports are determined by the *set of architectural registers*. The instruction set and set of architectural registers together describe a *computer architecture*. Note that the computer architecture does not describe how exactly instructions are executed or how long it takes to execute them; this is specific to a *microarchitecture* which implements the architecture. As an example, the Intel Core 2 Quad Q9550 is a microarchitecture which implements the AMD64 architecture.

Many microarchitectures implement an architecture and support additional instructions or registers through so-called *instruction-set extensions*. Examples are the AltiVec extensions found on many PowerPC processors and the Streaming SIMD Extensions (SSE) on x86 and AMD64 processors.

Most architectures considered in this thesis are *load-store architectures*. This means that arithmetic and logical instructions can operate only on values in registers. A typical computation first loads inputs from memory into registers, then performs arithmetic and finally stores the outputs back to memory. An important technique to make software run fast on such architectures is to avoid loads and stores by keeping values in registers as much as possible.

The opposite approach to a load-store architecture is a *memory-to-memory architecture* that does not have any registers. The only instruction arguments are memory addresses, and all instructions include loading inputs and storing outputs. This thesis does not consider pure memory-to-memory architectures but it does consider architectures (such as the x86 and the AMD64 architectures) that weaken the concept of a load-store architecture by allowing one input argument to be a memory address for some arithmetic instructions.

One important task in optimizing an assembly program is choosing the best instructions which together implement the targeted algorithm. Choosing the best instructions does not mean choosing the shortest sequence of instructions; the relation between the number of instructions and the resulting performance of a program is much more subtle. The reason for the complex relation between the choice of instructions and resulting execution speed lies mainly in the capabilities of modern processors to exploit parallelism on various levels.

2.2 Exploiting instruction-level parallelism

A computer program is nothing but a sequence of instructions. Some instructions in this sequence directly depend on each other but many programs have instructions with independent inputs and outputs; these instructions can in principle be swapped or executed in parallel. Parallelism from independent instructions is called *instruction-level parallelism*.

2.2.1 Dependencies between instructions

Before looking at how this instruction-level parallelism can be exploited to decrease execution time of a program, it is important to understand what it means for two instructions to be independent. One usually distinguishes three types of dependencies between instructions: *data dependencies*, *name dependencies*, and *control dependencies*, see, for example, [HP07, Section 2.1].

Data dependencies. If the output of an instruction i is input to an instruction j then there is a data dependency between these two instructions. Data dependencies are transitive, so when there is a data dependency between an instruction i and an instruction k and a dependency between instruction k and an instruction j , then there is also a data dependency between instructions i and j . Instructions with data dependencies cannot be swapped; the output of instruction i has to be computed before it can be used as input for instruction j .

Name dependencies. If an instruction j writes to a location that an earlier instruction i reads, there is no data flow between these two instructions, but they are not independent, i.e. they cannot be swapped. Similarly, if two instructions i and j write to the same location, they cannot be swapped although there is no data flow between the two instructions.

These dependencies are not true dependencies; they can be resolved by changing the output location of the second instruction if additional locations are available. The input locations of all subsequent instructions using the result have to be changed accordingly.

Control dependencies. Aside from arithmetic instructions, load instruction, and store instructions, programs contain *branch instructions* (also called jump instructions) that change the program flow, possibly depending on values known only at run-time. These instructions are required to implement loops and conditional statements. In a loop a conditional branch instruction at the end of the loop branches program execution to the beginning of the loop dependent on whether the loop condition holds. A conditional statement of the form

```
if c then A
else B
```

is realized in assembly by two branch instructions (here denoted **goto**) as follows

```
goto ELSECASE if not c
A
goto FINAL
ELSECASE:
B
FINAL:
```

Just as in this pseudocode, assembly programs use *labels* (here `ELSECASE` and `FINAL`) as placeholders for program addresses that are replaced by actual addresses before the program is executed. The decision bit *c* is usually a value in a register or a bit in a special *flag register*. Flags in the flag register are usually set as a side effect of certain arithmetic instructions.

Swapping an instruction with a control instruction or moving instructions across *branch targets* changes the semantics of a program. These dependencies of instructions on the control flow of a program are referred to as *control dependencies*.

2.2.2 Pipelined and superscalar execution

Two techniques enable modern processors to exploit instruction-level parallelism: pipelining and superscalar execution.

Pipelined execution. The idea of pipelining is based on the observation that execution of one instruction involves multiple steps: the instruction is first loaded from memory, then it is decoded and inputs are retrieved, the instruction is executed, for load and store instructions memory access is performed at the addresses computed before, and finally the result is written. For independent instructions these steps can overlap; while one instruction is decoded, the next instruction can already be loaded from memory and during execution of one instruction the next instruction can be decoded and so on. This overlapping in the execution of independent instructions is called pipelining. Note that some stages in the pipeline can also be overlapped for

dependent instructions as long as the dependency is not a control dependency. The time it takes for an instruction to move from one pipeline stage to the next pipeline stage is defined in [HP07, A.1] as processor cycle which may be different from a clock cycle. In fact different parts of the CPU can operate at different speeds; for example, on the Intel Pentium 4 processor two ALUs operate at double speed [Int03]. Throughout this thesis a *cycle* is a cycle as given by the processor manufacturer; for example a 2.4 GHz Intel Core 2 processor runs at a speed of 2.4 billion cycles per second. On Linux systems the exact *CPU frequency* in cycles/second can be obtained from `/proc/cpuinfo`.

The above-described steps or *pipeline stages* are typical for the pipeline of a simple reduced-instruction-set (RISC) processor, see e.g., [HP07, Section A.1]. Many processors decompose the execution of one instruction into many more pipeline stages and thereby achieve a much higher *pipeline depth*. For example, the Intel Pentium 4 processors of the Prescott family have a 31-stage pipeline, although not all instructions have to pass all stages. More pipeline stages have the advantage that the CPU needs to do less work in each pipeline stage which allows an increase in the CPU frequency. For details see for example [HP07, Section A.1].

Superscalar processors. The second concept to exploit instruction-level parallelism is duplicating certain parts of a processor to handle multiple instructions in the same pipeline stage in the same cycle. For example duplicating the *arithmetic-logic unit (ALU)* allows handling multiple instructions in the execution stage of the pipeline. Of course this only makes sense if also other units of the processor are duplicated. Processors that can issue multiple instructions per cycle are called superscalar processors. It is very common to say that some processor can, for example, “execute up to 3 integer-arithmetic instructions per cycle”, or “1 load/store instruction per cycle”. Such statements usually ignore possible bottlenecks in many pipeline stages, they should be understood as an upper bound derived from the number of ALUs or load-store units in the superscalar design of the processor.

2.2.3 Optimizing code for in-order CPUs

Many processors execute instructions in the same order they appear in the program. This is called *in-order execution*. To achieve optimal speed the programmer not only needs to choose appropriate instructions, a crucial task to benefit from pipelining and superscalar execution is *instruction scheduling*. The most important target when scheduling instructions is to hide latencies of instructions by placing independent instructions between an instruction producing a result and the first instruction using the result. In order to avoid name dependencies, all these independent instructions need to write to different output locations (usually registers). Choosing registers for values involved in the computation is called *register allocation*. If the set of architectural registers is not large enough to hold all required values, some values need to be stored to memory and loaded back to a register later; this is called a *register spill*. On most microarchitectures spills take additional cycles, so register allocation generally tries to keep the number of spills as low as possible.

Note that instruction scheduling and register allocation have opposing requirements on the use of registers. Interleaving independent instructions requires more registers but using more registers than available in the set of architectural registers requires spills.

Aside from careful instruction scheduling and register allocation, some in-order microarchitectures require additional conditions to be fulfilled to achieve optimal performance. Such conditions include, for example, instruction alignment as described in Chapter 4 for the Synergistic Processor Units (SPUs) of the Cell Broadband Engine or instruction grouping on the UltraSPARC II processor which for example needs to make sure that a shift instruction is the first of two integer-arithmetic instructions executed in one cycle.

2.2.4 Optimizing code for out-of-order CPUs

Many modern processors, in particular most Intel and AMD processors, execute the instructions in a program out of order, the processor dynamically schedules instructions at run time. The main advantage of *out-of-order execution* is that programs can be compiled once and achieve reasonable performance on different microarchitectures. Furthermore it releases some pressure from the compiler, it becomes easier and cheaper to develop compilers for new microarchitectures. Finally the processor can schedule instructions based on information which is known only at run time and not at compile time; some data dependencies for example may depend on input-dependent memory addresses.

One important feature of out-of-order processors is that their register file usually contains many more registers than their set of architectural registers. The larger set of *physical registers* is used to resolve name dependencies in hardware by dynamically assigning architectural registers to physical destination registers. This process is known as *register renaming*. For more details on register-renaming techniques see for example [HP07, Section 2.4].

Carefully optimized software implemented in assembly usually does not benefit from out-of-order execution. Quite the contrary: the effects of instruction scheduling on execution speed are much harder to predict, also because manufacturers often do not document how exactly out-of-order execution is implemented. For many Intel and AMD processors a good source of information is the manuals by Fog [Fog10c, Fog10b, Fog10a].

2.3 Exploiting data-level parallelism

Whenever the same computations need to be carried out on multiple independent inputs, these computations can in principle be carried out in parallel. This kind of parallelism is called *data-level parallelism*. Carrying out the same computation on independent data is often referred to as *batching* of computations.

The best way to exploit data-level parallelism to speed up execution of a program depends on

- how many instructions the computations on independent data consist of, ranging from a single arithmetic operation to a large function or program;
- the degree of data-level parallelism, i.e., the number of independent inputs, this can range from just two up to (potentially) billions of independent inputs;
- the computer architecture; and
- what computations are carried out.

There are several techniques to exploit data-level parallelism, as described in the following subsections. With a sufficient degree of data-level parallelism these techniques can be combined to achieve best performance.

2.3.1 Reducing to instruction-level parallelism

A simple example for data-level parallelism is the addition of two 4-component 32-bit integer vectors. This operation requires loading the 2×4 inputs, performing 4 integer additions and storing the 4 results. This can be done in 8 load instructions, 4 integer-addition instructions and 4 store instructions; most of these instructions are independent and the resulting instruction sequence has a high degree of instruction-level parallelism which can be exploited using the techniques described above.

Performing the same operations on independent data streams can always be translated into a program with a high degree of instruction-level parallelism.

2.3.2 Single instruction stream, multiple data streams (SIMD)

Flynn in [Fly66] categorizes “very high-speed computers” as follows:

Very high speed computers may be classified as follows:

1. Single Instruction Stream—Single Data Stream (SISD)
2. Single Instruction Stream—Multiple Data Stream (SIMD)
3. Multiple Instruction Stream—Single Data Stream (MISD)
4. Multiple Instruction Stream—Multiple Data Stream (MIMD).

“Stream” as used here, refers to the sequence of data or instructions as seen by the machine during the execution of the program.

Instead of trying to fit each computer architecture into one of these classes, they can instead be understood as computational paradigms which an architecture can implement. This understanding of the terms meets the reality of modern processors which in fact often implement at least two of the paradigms.

The SISD paradigm is what this chapter considered so far, executing a single sequence of instructions (the program) on a single stream of data. While the MISD paradigm has never been implemented in commercial computers, the SIMD and MIMD paradigms become more and more important in modern processors. The SIMD

paradigm directly exploits data-level parallelism. The two implementations of SIMD found in current processors are *vector registers* and *single instruction, multiple threads (SIMT)*.

Vector registers. The idea of vector registers is to keep multiple values of the same type in one register. For example a 128-bit vector register can keep four 32-bit integers or two 64-bit (double-precision) floating point values. Arithmetic operations are then carried out on all of these values in parallel by *vector instructions* (sometimes called SIMD instructions).

Many architectures support instructions on vectors of different data types, typically 8-bit, 16-bit and 32-bit integers, and single-precision and double-precision floating-point values. Examples are the Streaming SIMD Extensions of x86 processors [TH99] which became part of the AMD64 instruction set, the AltiVec extension of various PowerPC processors [Fre], and the instruction set of the Synergistic Processor Units of the Cell Broadband Engine [Son06].

Coming back to the example of the addition of two 4-component 32-bit integer vectors: This can be carried out using vector instructions on 128-bit registers using two 128-bit load instructions, one 32-bit-vector-addition instruction and one 128-bit store instruction. Note that the vector instructions of all architectures described in the following chapters can load multiple values from memory into a vector register only if they are stored consecutively. Loading, for example, four 32-bit integers from four non-consecutive positions in memory into one 128-bit register in one instruction is not possible. In fact, collecting values in a vector register which are stored at non-consecutive memory positions often requires many instructions.

Note that any n -bit register that supports bit-logical operations can be seen as a vector register containing n 1-bit values. This observation is important for “bitslicing”; Chapters 3 and 6 will present examples of this technique.

Single instruction multiple threads (SIMT). Many modern graphics processing units (GPUs) implement SIMD by executing the same instruction in parallel by many hardware threads (see also Section 2.4). The program is the same for all threads. Accessing different input data is realized by loads from addresses that depend on a thread identifier.

The main difference compared to the concept of vector registers is handling of memory loads: Unlike current implementations of vector registers, SIMT makes it possible to let all threads load values from arbitrary memory positions in the same instruction. Note that this corresponds to collecting values from arbitrary memory positions in a vector register in one instruction. However, for current implementations of SIMT the performance of such a load operation depends on the memory positions the threads load from; this issue will be discussed in detail in Chapter 6.

Note that even with a high level of data-level parallelism, making the best use of the SIMD capabilities of an architecture may require different algorithmic approaches and changes to the data representation. The implementations described in Chapters 3, 4, and 6 are examples for what algorithmic changes are required to exploit the computational power of an SIMD instruction set.

2.3.3 Algorithmic improvements

Both the translation to instruction-level parallelism and SIMD duplicate a single-input computation and benefit from hardware which accelerates this computation. Sometimes it is also possible to benefit from data-level parallelism by using an algorithm different from the one typically used for the single-input case. Such algorithmic improvements are very different from the other, computer-architecture-related, techniques described in this chapter; nevertheless they should be mentioned in the context of data-level parallelism because they are highly relevant in the context of many cryptographic and cryptanalytic computations.

An example for such algorithmic changes is the computation of inverses in a field: Let K be a field and $a, b \in K$. Computing a^{-1}, b^{-1} can be done by first computing ab , then using one inversion to obtain $(ab)^{-1}$ and then using one multiplication with a to obtain b^{-1} and one multiplication with b to obtain a^{-1} . This algorithm is known as Montgomery inversion and can be generalized to inverting n values using $3(n-1)$ multiplications and 1 inversion [Mon87]. Multiplications can be computed much more efficiently than inversion in many fields; if 3 multiplications are more efficient than one inversion this algorithm becomes more efficient the larger the number of inputs is. Another example for such algorithmic improvements for batched computations in cryptography are multi-exponentiation algorithms. See, for example, [ACD⁺06, Section 9.1.5].

2.4 Exploiting task-level parallelism

Larger software often involves many partially independent tasks which only occasionally need to exchange information. Many processors implement the MIMD paradigm, and support parallel execution of many tasks for example through multi-core processors. Such processors combine multiple processors (called processor *cores*) on one chip. Each core completely implements an architecture with all registers and units; some resources, in particular caches (see Section 2.6.1), are usually shared between the cores. For example, an Intel Core 2 Quad Q6600 CPU contains 4 AMD64 processor cores that can be used, as if they were separate processors. Usually, but not necessarily, all cores of one processor implement the same architecture; a counterexample is the Cell Broadband Engine which combines cores of two different architectures on one chip.

Exploiting such MIMD capabilities requires that the programmer implements different tasks as independent programs or as one program with multiple *threads*. Threads are independent subprograms of one program that can—unlike independent programs—access the same address space in memory, which can be used for efficient communication between threads. Beware that communication through shared memory (or generally access to a shared resource by multiple threads or programs) raises synchronization issues. These issues are relevant in the implementations presented in Chapters 6 and 7. A detailed discussion is omitted here; for an introduction to multi-threaded programming see for example [HP07, Chapter 4].

Most cryptographic software is small and does not involve independent tasks, but it is sometimes possible to use MIMD capabilities of the processor for SIMD computations. For example, encryption of many gigabytes of data can be done by multiple threads or programs, each encrypting a part of the data.

The exploitation of task-level parallelism becomes much more important for cryptanalytical software. For many cryptanalytical applications (such as the one presented in Chapter 6) the MIMD capabilities of processors and computer clusters are used for SIMD computations: The same computation is carried out on different inputs on multiple cores of multiple computers without communication between them. The software presented in Chapter 7 shows more advanced use of MIMD capabilities involving communication between threads running on one computer and also between the multiple computers of a cluster.

2.5 Function calls, loops, and conditional statements

Frequent branches to other positions in a program can seriously degrade the performance of the program. The most common reasons for branches are function calls, loops and conditional statements. This section explains how these branches influence performance and describes techniques to avoid them.

Function calls and inlining. A function at the assembly level is a part of a program which is defined by an entry point and a return point. Functions are used from other parts of the code through *function calls* which store the address of the current instruction, and then branch to the entry point of the function. At the return point of the function, execution is continued at the stored instruction address from which the function was called. One advantage of functions is that the same piece of code can be reused; furthermore, a function can be implemented without knowing the context from which it is called. A function can be implemented in a separate file that is translated to a separate machine-language object file. One implication of this flexible concept is that it needs to be specified how input arguments are passed from the *caller* to the *callee* and how a return value is passed to the *caller*. Furthermore it has to be assured that a function does not overwrite values in registers that are still needed in the code from which the function is called. For compiled programming languages such as C or C++ these details of function calls are specified in a function-call convention. Programs implemented entirely in assembly can use their own function-call convention, while functions implemented in assembly which are called from, for example, a C program need to respect the existing convention. Typically a function call involves the following:

- saving *callee registers* (registers that the called function is free to use) by the caller,
- moving the stack pointer to reserve space on the stack for local variables of the function,

- moving arguments to input registers specified in the calling convention (for arguments that are passed in registers),
- placing arguments on the stack (for arguments that are not passed in registers),
- branching to the function entry point,
- saving *caller registers* inside the function if the function requires registers that potentially contain values still required by the caller, and
- loading stack arguments into registers.

Returning from the function involves the inverse steps (restoring caller registers, passing the return value, moving back the stack pointer, restoring callee registers). Clearly the call to a function involves a significant overhead to the computations carried out inside the function.

The standard way to remove this overhead is *function inlining*, i.e. replacing each function call with the instructions belonging to the function and removing overhead which is not required in the current context. Note that this technique increases the size of the program, at least if the function is used more than once; increasing code size can also degrade performance as will be described in Section 2.6.

Loops and loop unrolling. How loops influence performance is best illustrated using an example.

Consider a loop which increases each element of an array of 32-bit integers of length 1024 by 1, hence in C notation:

```
for(i=0;i<1024;i++) a[i]+=1;
```

This loop can be implemented in assembly using 6 instructions:

- load `a[i]` into a register,
- increase the value in the register by 1,
- store the value back to `a[i]`,
- increase `i` by 1,
- check whether `i` is smaller than 1024, and
- branch back to the beginning of the loop depending on the result of the previous check.

Implementing the loop this way is bad for performance for multiple reasons. The first reason is that loop control contributes a 100% overhead to the required 3 instructions per array element. The second reason is that the loop body can not hide instruction latencies: There are data dependencies between the first three instructions and between the last three instructions. The third reason is the effect of conditional branch instructions on pipelined execution: At the point where the condition

of the branch instruction is evaluated, several instructions have already entered earlier pipeline stages. If these instructions belong to the path which is taken after the branch, this is not a problem. However if the instructions belong to the path not taken they are useless for the computation, the pipeline has to be flushed and the instructions from the other path have to be fed to the pipeline which usually takes several cycles. In order to determine which instructions should be fed to the pipeline at a conditional branch instruction, most modern processors use sophisticated *branch-prediction* techniques, for details see for example [HP07, Section 2.3].

To illustrate the influence of these effects on performance consider the Synergistic Processor Units of the Cell Broadband Engine. All instructions operate on 128-bit vector registers, each iteration of the loop can thus process 4 32-bit integers in parallel. The 128-bit load instruction has a latency of 6 cycles, the 4-way SIMD 32-bit integer addition has a latency of 2 cycles. The first 3 instructions therefore take at least 9 cycles. Incrementing the counter and the comparison have a latency of 2 cycles each. These instructions can be carried out while the addition waits for the result from the load instruction and thus do not take additional cycles. The final branch instruction takes in the best case one cycle. At least one of the branches will be mispredicted incurring an additional penalty of 18 to 19 cycles. In total the required 256 iterations of the loop take $256 \cdot 10 + 18 = 2578$ cycles in the best case.

A much more efficient way to implement this loop is through *loop unrolling*. This means that all loop-control instructions are removed and the 256 iterations are implemented in 256 load instructions, 256 additions and 256 store instructions. The SPU can carry out an addition and a load or store instruction in the same cycle so with careful instruction scheduling the unrolled loops takes 512 cycles for 256 load and 256 store instructions. The 6 cycles latency of the first load instruction and the 2 cycles latency of the final addition increase this count by only 8. The complete unrolled loop takes only 520 cycles, a performance increase by almost a factor of 5.

The disadvantage of this approach (and loop unrolling in general) is the increase of code size. The loop in the example takes 6 instructions, the unrolled version requires 768 instructions, an increase by a factor of 128. A compromise is partial unrolling: For example on the SPU this loop could be implemented using 8 load instructions and 2 additions in an initial computation. The additions can be interleaved with the final 2 loads so this phase takes only 8 cycles. The loop then consists of 6 loads, 6 additions and 6 stores. Additions use values loaded in the previous loop iteration or—for the first iteration—in the precomputation. Each loop iteration stores values computed in the additions in the previous loop iteration. In each iteration additions and instructions to increase the loop counter and the comparison can be executed in parallel with the loads and stores. Including the final branch instruction each iteration takes 13 cycles, assuming correct branch prediction. After 41 iterations of the loop there are 2 load instructions, 8 additions and 10 stores remaining which take additional 14 cycles. In total this partially unrolled loop takes $8 + 41 \cdot 13 + 14 + 18 = 573$ cycles including an 18-cycle penalty for one mispredicted branch. This is slightly slower than the fully unrolled loop but this version only requires 57 instructions.

Note that loop unrolling works best for loops with a fixed number of iterations; partial unrolling can also be done for loops with a variable number of iterations but involves some additional instructions to handle the case of a final non-complete iteration.

Conditional statements. Conditional statements implemented with branch instructions incur the same performance penalties from branch mispredictions as conditional branch instructions in loops. Aside from these performance issues they can also breach security of cryptographic applications: If the condition of a conditional branch instruction depends on secret input, the program will take different time to execute depending on this condition and thus depending on secret data. This opens up an opportunity for a so-called *timing attack*; an attacker can deduce information about secret data from measurements of the execution time of the program.

Constant-time implementations of cryptographic primitives and protocols are programs whose execution time does not depend on secret input. In constant-time software conditional statements can be implemented through arithmetic operations: Let for example b , x , and y be integer variables and let b have value either 1 (true) or 0 (false). The conditional statement

if b then $x \leftarrow y$

can be evaluated with arithmetic operations as

$$x \leftarrow by + (1 - b)x.$$

2.6 Accessing memory

Accessing memory is a central part of almost every program and many aspects of how exactly memory and access to memory is implemented influence the performance of programs.

All microarchitectures considered in this thesis employ small but fast storage between registers and main memory to accelerate access to frequently-used data. On some microarchitectures this storage has to be used explicitly by the programmer; an example is the local storage of the Synergistic Processor Units of the Cell Broadband Engine. Other architectures transparently store data in such small and fast storage when it is read from or written to main memory. Such transparent fast storage is called *CPU cache* or simply *cache* if the context is clear. Aside from data, also program instructions need to be retrieved from memory; loads of instructions are also cached. A cache which is exclusively used for instructions is called *instruction cache* as opposed to a *data cache* which is used exclusively for data.

2.6.1 Caching

A processor with cached access to memory loads data by first checking whether the data is stored in the cache. If this is the case (*cache hit*) data is retrieved from the

cache typically taking only a few cycles. Otherwise (*cache miss*) data is retrieved from main memory which typically has a latency of several hundred cycles.

Data is loaded into the cache in fixed-size units called *cache lines*. Each cache entry carries a tag with the address of the corresponding cache line in main memory; this tag is used to determine whether data from a certain memory address is in cache or not. When the data requested by a load instruction is not in cache (*read miss*), the whole cache line worth of data containing the requested data is fetched from memory into cache. Loading data which crosses the boundary of a cache line in memory either takes significantly longer or results in an error, depending on the architecture. At what position a cache line is placed in cache (and thus which previously cached data is replaced) depends on two implementation details of cache: associativity and replacement policy.

Cache associativity. One strategy to decide where a cache line is placed in cache is to map the address of each cache line in memory to exactly one position in cache, usually the cache-line address modulo the number of cache lines of the cache. This assignment of addresses to a cache position is called *direct mapping*. Another way of assigning cache-line addresses to cache positions is to partition the cache positions into multiple sets, map each address to one of these sets and place the cache line at some position within this set. Caches using this scheme are called *n-way set associative* where *n* is the number of cache lines per set. A cache with only one set, i.e. a cache where each cache line can be placed at any position, is called a fully associative cache; such caches cannot be found in real-world computers. Note that a direct mapped cache can be seen as a 1-way set-associative cache.

Cache replacement strategy. For all but direct mapped caches a cache line has different possible positions in cache and there are different strategies to determine at which positions it is placed and thus which previously cached data is replaced. A very common policy is to replace the least-recently used (LRU) cache line. A true LRU policy is expensive to realize in hardware, in particular for highly associative caches. Most modern processors therefore use a pseudo-LRU policy that approximates LRU behavior. Other policies include choosing a random position within the set, using a round-robin approach or replacing the oldest cache line (first-in, first-out).

The situation of a read access to the cache considered so far is easier than behavior of the cache on a write access to memory. For a write-access cache hit the data can either be written to cache and to memory; this is called a write-through cache. Another possibility is to write only to the cache and write to memory only when the cache line is replaced; this is called a write-back cache. A write miss can be handled by either first fetching the corresponding cache line to cache as for a read miss and then continuing as in the case of a cache hit, or by writing data to memory without modifying the cache.

Most processors which support cached access to memory have several levels of cache. The smallest and fastest is called level-1 cache. A typical size for the level-1 cache are a few KB; for example, the Intel Core 2 processor has a 32 KB level-1 data cache. On multi-core processors each core usually has its own level-1 cache; higher-level caches (for example between 2 and 12 MB of level-2 cache for Intel Core 2

processors) are often shared between the cores. Higher-level caches are also often used for both data and instructions whereas level-1 data and instruction caches are usually separated.

For cryptographic applications effects of caching not only influence performance but also security. Just as conditional branches can influence the execution time of a program depending on secret inputs, any load from an address that depends on secret data can take a different number of cycles, depending on whether the data at this address is in cache or not. In fact timing variation due to loads from cached memory is much more subtle, and involves many microarchitecture-dependent effects; for details see [Ber04a, Sections 10–15]. The most convincing way to make implementations secure against *cache-timing attacks* is to avoid all loads from addresses that depend on secret inputs. Chapter 3 will discuss cache-timing attacks and implementation techniques to avoid them in the context of the Advanced Encryption Standard.

Many cryptographic (and also cryptanalytical) computations involve only small amounts of frequently used data which then completely fit into the level-1 cache of all modern processors. For some algorithms (and microarchitectures) it is still important to know the details about the associativity of the cache to align data in memory in a way that prevents frequently used data from being replaced in cache; this also includes alignment of the stack. More serious than performance penalties due to data-cache misses are penalties due to instruction-cache misses: For implementations of complex cryptographic primitives (such as, e.g., pairings, considered in Chapter 5) extensive use of implementation techniques as function inlining and loop unrolling easily blows up the code size far beyond the size of the level-1 cache leading to serious performance degradation.

2.6.2 Virtual-address translation

All modern operating systems for general-purpose computers use the concept of virtual memory. This means that addresses used by a program to load and store data in memory are only *virtual memory addresses*. These addresses need to be translated to *physical addresses* for each memory access. One advantage of virtual memory is that memory fragmentation is hidden from programs: they can use continuous memory addresses even if not enough continuous space is available in physical memory. Furthermore it is possible to offer a larger amount of virtual memory to programs than physical memory is available. Some of the virtual memory addresses are then mapped to space on the hard disk and are swapped in and out of memory as required.

The mapping between virtual addresses and physical addresses is done by partitioning the virtual memory space into *memory pages* and managing a table that maps each memory page to a location either in physical memory or on the hard disk. This table is called the *page table*, and is usually accessed through a dedicated cache called the *translation lookaside buffer (TLB)*.

There are various ways that virtual-address translation influences the performance of programs. A very serious penalty is for example incurred by a *page miss*, a request for a memory page which is not located in physical memory but on the hard

disk and first needs to be copied into memory. Other effects relate to the question whether cache tags and indices are using virtual or physical addresses and to cache hits and misses when looking up physical addresses in the TLB. For more details see for example [HP07, Section C.4]. For the software described in this thesis effects of virtual-address translation are small.

2.7 Limitations of compilers

Compilers which translate from a high-level language to machine language usually do not generate code running at a similar speed as hand-optimized assembly code. There are two main reasons why this is the case:

The first reason is that compilers are not optimized to compile code which aims at high performance but for “typical” code. This means for example that an important measure to evaluate the quality of a compiler is the time it takes to generate code, not only the performance of the resulting code. Furthermore several problems in code generation, most prominently register allocation and instruction scheduling are NP-complete and tackled by compilers through heuristics that perform well in the average case. Many cryptographic programs are not average cases but best cases with respect to these problems. For example register allocation can be solved in linear time for any fixed number of registers for so-called “structured” programs if a register assignment without spills exists [BGT98]; heuristic algorithms tuned for the average case may not find this optimal solution.

The second reason is that abstraction of high-level languages makes it impossible to access certain features of a specific processor. For example the C programming language supports addition of two 64-bit integers (`type long long`). However the carry bit of this addition is lost, the result has only 64 bits. Many 64-bit architectures, for example the AMD64 architecture, support addition of two 64-bit integers where the carry bit is saved in a special *flag register*. It can then be used in a subsequent addition or in a conditional statement.

2.8 The qasm programming language

Writing software in assembly which runs faster than for example a well-written and compiler-optimized C program takes a lot more time than implementing in, for example, C. In order to reduce the programming time and make the development of high-speed software on assembly level more efficient, Bernstein developed the qasm programming language [Ber]. The idea of this language is to maintain all possibilities and control of assembly programming but support the programmer with a register allocator and unified syntax:

Register allocation. In assembly programs it is the programmer’s task to assign registers to “variables”; even if all live variables (variables whose values are still required) fit into the architectural registers throughout the program this can be a tedious task. Programs implemented in qasm can declare an arbitrary number of *register variables*

for each type of register. These will be assigned to registers automatically by a register allocator if enough registers are available. If not enough registers are available, no spill code will be generated by the register allocator, the task of deciding which variables to spill is still left to the programmer. Unlike a variable declared with the keyword `register` in C, register variables in `qasm` are thus guaranteed to be in registers. In most contexts it is not important in which register a certain value is stored; a variable of type `int64` can usually be assigned to any 64-bit integer register. However, in some situations it is important to have control about what register is assigned to a variable at a certain point. For this case `qasm` supports statements that instruct the register allocator to assign a certain variable to a specific register – no control is lost compared to a program written in assembly.

Unified syntax. Programming assembly on various different platforms comes with an additional inconvenience: assembly syntax is different (and not at all intuitive) on different architectures. For example shifting the content of a 64-bit integer register right by 5 bits is done as

```
srlx %i1,5,%i1
```

for the UltraSPARC architecture, as

```
srldi 1,1,5
```

for the PowerPC64 architecture, and as

```
shr $5,%rsi
```

for the AMD64 architecture. In these examples, `%i1`, `1`, and `%rsi` are names of registers. Note that not only the instruction and the register names, but also the order of arguments and the syntax for immediate values are different.

The `qasm` programming language unifies syntax in the following sense: If on two different architectures there exist instructions which do the same, then they have the same syntax in `qasm`. In the case of the above example this syntax is

```
r = (uint64) r >> 5
```

where `r` is a 64-bit integer register variable. Unifying syntax does *not* mean that the same set of instructions exists on different architectures; there exists a one-to-one map between `qasm` instructions and assembly instructions for each architecture. This map from `qasm` instructions to assembly instructions is specified in *machine-description* files, one file per architecture. Instructions are specified one per line, a typical line looks as follows:

```
r = (uint64) s >> n:>r=int64:<s=int64:#n:asm/srlx <s,#n,>r:
```

This line is taken from the UltraSPARC machine-description file and specifies an instruction of the form

```
r = (uint64) s >>= n
```

where `r` is an output register of type `int64`, `s` is an input register of type `int64`, and `n` is an immediate value. This instruction is mapped to the `srlx` instruction, with the first argument being the register currently assigned to the variable `s`, the second argument being the immediate value `n` and the third argument being the register currently assigned to the variable `r`.

The syntax of `qasm` tries to stay as close as possible to C syntax; as each line contains exactly one instruction (which is mapped to exactly one assembly instruction), semicolons are omitted to separate instructions. Most software presented in this thesis is implemented in `qasm`, some code examples are also given in `qasm` syntax, in particular when explaining general assembly concepts. Other examples use assembly syntax, in particular when explaining concepts for a specific architecture or if the semantics of the corresponding `qasm` instructions are not clear without further explanation.

3

Implementations of the Advanced Encryption Standard

In September 1997 the National Institute of Standards and Technology (NIST) issued a public call for proposals for a new block cipher to succeed the Data Encryption Standard (DES) [NIS79]. Out of 15 submitted algorithms the Rijndael cipher by Daemen and Rijmen [DR99] was chosen to become the new Advanced Encryption Standard (AES) in November 2001 [NIS01]. This chapter describes implementation techniques for AES, which is now one of the most widely used symmetric cryptographic algorithms.

Section 3.1 describes the Advanced Encryption Standard and the Galois Counter Mode of operation which was proposed by McGrew and Viega in 2004 [MV04] and standardized by NIST in 2007 [Dwo07] to provide authenticated encryption. Section 3.2 consider a class of attacks against implementations of AES—so-called cache-timing attacks—that have been successfully mounted against lookup-table-based implementations of AES. The main part of the chapter is devoted to implementation techniques (in Sections 3.3 and 3.4) and benchmarks (in Section 3.6) of AES and AES-GCM on different architectures. All the implementations of AES described in this chapter are compatible with the eSTREAM API [Can08]. AES in counter mode is compatible with the eSTREAM reference implementation.

This chapter is based on joint work with Bernstein published in [BS08] and with Käsper published in [KS09]. Aside from some minor changes to notation and phrasing the main differences between the content of this chapter and [BS08] and [KS09] are the following:

- the background on AES and AES-GCM in Section 3.1 is more detailed than the corresponding sections in [BS08] and [KS09];

- the benchmarks in Section 3.6 are more detailed and use different machines; and
- the comparison of the AES-GCM performance with LibTomCrypt, which is contained in [KS09] is omitted here since the performance results of LibTomCrypt are not online anymore, instead the results are compared with recent benchmarks of AES-GCM in the Crypto++ library [Dai09a].

3.1 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard is a block cipher with a fixed block length of 128 bits. It supports three different key lengths: 128 bits, 192 bits, and 256 bits. Encrypting a 128-bit block means transforming it in n rounds into a 128-bit output block. This transformation involves $n + 1$ round keys K_0, \dots, K_n derived from the AES key in a so-called key expansion. The number of rounds n depends on the key length: $n = 10$ for 128-bit keys, $n = 12$ for 192-bit keys, and $n = 14$ for 256-bit keys. The 16-byte input block $(b_0, \dots, b_9, b_a, \dots, b_f)$ which is transformed during encryption is usually written as a 4×4 byte matrix

$$B = \begin{pmatrix} b_0 & b_4 & b_8 & b_c \\ b_1 & b_5 & b_9 & b_d \\ b_2 & b_6 & b_a & b_e \\ b_3 & b_7 & b_b & b_f \end{pmatrix} = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix},$$

the so-called AES state.

The following description considers encryption and key expansion of the 128-bit variant of AES; the 192-bit and 256-bit variants involve a slightly different key expansion, but the operations in each round are the same for all variants. The implementations described in this chapter perform AES in the counter (CTR) mode of operation, which does not require AES decryption.

3.1.1 AES encryption

The encryption of a 128-bit (16-byte) block of input is performed as described in Algorithm 1. This description uses four basic operations, SUBBYTES, SHIFTROWS, MIXCOLUMNS, and ADDROUNDKEY described in detail in the following.

ADDROUNDKEY(B, K). The ADDROUNDKEY operation returns the bitwise xor of the AES state B and K .

SUBBYTES(B). The SUBBYTES operation substitutes each byte of B by another byte. The input to the SUBBYTES operation can be a byte array of arbitrary length – for encryption the input consists of the 16 bytes of the AES state, in the key expansion the input has only 4 bytes (see Subsection 3.1.3). The byte substitution is based on the following algebraic construction: Consider each byte b of B as an element of

Algorithm 1 AES-128 encryption**Input:** 128-bit input block B , 128-bit AES round keys K_0, \dots, K_{10} **Output:** 128-bit block of encrypted output $B \leftarrow \text{ADDROUNDKEY}(B, K_0)$ **for** i from 1 to 9 **do** $B \leftarrow \text{SUBBYTES}(B)$ $B \leftarrow \text{SHIFTRows}(B)$ $B \leftarrow \text{MixCOLUMNS}(B)$ $B \leftarrow \text{ADDROUNDKEY}(B, K_i)$ **end for** $B \leftarrow \text{SUBBYTES}(B)$ $B \leftarrow \text{SHIFTRows}(B)$ $B \leftarrow \text{ADDROUNDKEY}(B, K_{10})$ **return** B

$\mathbb{F}_{2^8} = \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)$ and compute the inverse $y = b^{-1}$ (for $b = 0$ set $y = 0$). Let y_0, \dots, y_7 denote the binary coefficients of y and apply the following affine map to obtain the result of the byte substitution r :

$$r = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

Note that this byte substitution can easily be implemented through a 256-byte lookup table. However, Section 3.4 shows that the algebraic structure of the SUBBYTES operation is helpful for implementations not only on memory-restricted devices.

SHIFTRows(B). The SHIFTRows operation rotates the rows of the AES state by different distances:

$$\begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix} \mapsto \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{11} & b_{12} & b_{13} & b_{10} \\ b_{22} & b_{23} & b_{20} & b_{21} \\ b_{33} & b_{30} & b_{31} & b_{32} \end{pmatrix}.$$

MixCOLUMNS(B). Like the SUBBYTES operation, the MixCOLUMNS operation considers the bytes of the AES state as elements of $\mathbb{F}_{2^8} = \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)$. It multiplies the AES state matrix B with a fixed circulant matrix:

$$B \mapsto \begin{pmatrix} X & X+1 & 1 & 1 \\ 1 & X & X+1 & 1 \\ 1 & 1 & X & X+1 \\ X+1 & 1 & 1 & X \end{pmatrix} \cdot B.$$

3.1.2 Combining SUBBYTES, SHIFTRows, and MIXCOLUMNS

In the AES proposal Daemen and Rijmen suggest a speedup on processors with word length 32 or above [DR99, Section 5.2]. The idea is to not only use a lookup table for the SUBBYTES operation but to combine the SUBBYTES, the SHIFTRows, and the MIXCOLUMNS operations into 4 lookup tables T_0, \dots, T_3 of size 1 KB each. Each round transformation then requires 4 32-bit lookups from each of these tables, in total 16 table lookups.

The Tables T_1, T_2 and T_3 can be obtained from T_0 by rotating the 32-bit entries; the entries of table T_1 are obtained by rotating the entries of T_0 by one byte to the right, the entries of table T_2 are obtained by rotating the entries of T_0 by two bytes, and the entries of table T_3 are obtained by rotating the entries of T_0 by three bytes to the right.

3.1.3 AES key expansion

The AES key expansion algorithm is 32-bit-word oriented. The 128-bit AES key is written as 4 words W_0, \dots, W_3 ; from these words the key expansion generates an array of 44 words W_0, \dots, W_{43} as described in Algorithm 2. The round keys K_0, \dots, K_{10} are obtained as $K_i = (W_{4i}, W_{4i+1}, W_{4i+2}, W_{4i+3})$.

The key expansion algorithm makes use of the SUBBYTES operation which is also used in the AES encryption. Furthermore it uses a cyclic left shift by one byte denoted as ROTBYTE and constants $\text{RCON}_i, i = 1, \dots, 10$, which are defined as follows:

- $\text{RCON}_i = (\text{RC}_i, 0x00, 0x00, 0x00)$, and
- RC_i the byte representing X^{i-1} in $\mathbb{F}_{2^8} = \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)$.

3.1.4 The Galois/Counter Mode (GCM)

The Galois/Counter mode of operation [MV04] is a combined mode for encryption and authentication. Encryption is performed using AES in counter (CTR) mode, the counter is extended to a 16-byte input block by a nonce. The main operation for authentication is multiplication in the finite field $\mathbb{F}_{2^{128}} = \mathbb{F}_2[X]/(X^{128} + X^7 + X^2 + X + 1)$ with a field element H derived from the secret key but not changing across message blocks. This multiplication considers bit strings of length 128 as elements of the field $\mathbb{F}_{2^{128}}$.

The specification of the Galois/Counter mode describes the possibility to encrypt only parts of the input but authenticate all input; both the description in Algorithm 3

Algorithm 2 AES-128 key expansion

Input: 128-bit AES key $K = (W_0, \dots, W_3)$ **Output:** AES round keys (K_0, \dots, K_{10})

```

for  $i$  from 4 to 43 do
     $t \leftarrow W_{i-1}$ 
    if  $i \bmod 4 = 0$  then
         $t \leftarrow \text{SUBBYTES}(\text{ROTBYTE}(t)) \oplus \text{RCON}_i$ 
    end if
     $W_i \leftarrow W_{i-4} \oplus t$ 
end for
return  $(K_0 = (W_0, \dots, W_3), \dots, K_{10} = (W_{40}, \dots, W_{43}))$ 

```

and the implementation described in Section 3.5 are limited to encrypting and authenticating the whole input, furthermore they only consider a 32-bit counter (and consequently a 96-bit nonce).

Algorithm 3 AES-GCM encryption and authentication; AES_K denotes AES encryption under key K , $\text{bitlen}(M)$ denotes the bit length of M without padding

Input: Plaintext $M = (m_0, \dots, m_n)$, with m_i 16-byte blocks (m_n possibly padded by zero-bits), $n < 2^{32} - 2$, 128-bit AES key K , 96-bit nonce N **Output:** Ciphertext $C = (c_0, \dots, c_n)$ and 16-byte authentication tag T

```

 $H \leftarrow \text{AES}_K(0)$ 
 $T \leftarrow 0$ 
for  $i$  from 0 to  $n$  do
    Let  $w_{i+1}$  denote the 32-bit big-endian integer representing  $i + 1$ 
     $c_i \leftarrow m_i \oplus \text{AES}_K(N|w_{i+1})$ 
     $T \leftarrow (T \oplus c_i) \cdot H$ 
end for
 $T \leftarrow (T \oplus \text{bitlen}(M)) \cdot H$ 
 $T \leftarrow T \oplus \text{AES}_K(N|0x00000001)$ 
return  $((c_0, \dots, c_n), T)$ 

```

3.2 Cache-timing attacks against AES and GCM

Although related-key attacks against the key expansion of AES have lowered the security level of variants of AES in certain attack scenarios [BKN09, BK09], AES is still believed to be a very secure cryptographic algorithm. However, *implementations* of AES may be vulnerable to side-channel attacks; software implementations on general-purpose CPUs are mainly vulnerable to so-called cache-timing attacks.

Cache-timing attacks exploit the timing variability of data loads from memory. This variability is due to the fact that all modern processors use a hierarchy of caches to reduce load latency. If a load operation can retrieve data from one of the caches

(*cache hit*), the load takes less time than if the data has to be retrieved from RAM (*cache miss*), see also Subsection 2.6.1.

Kocher was the first to suggest cache-timing attacks against cryptographic algorithms that load data from positions that depend on secret information [Koc96]. Initially, timing attacks were mainly mentioned in the context of public-key algorithms until Kelsey et al. [KSWH00] and Page [Pag02] considered timing attacks, including cache-timing attacks, against secret-key algorithms. Tsunoo et al. demonstrated the practical feasibility of cache-timing attacks against the symmetric-key ciphers MISTY1 [TTMM02] and DES [TSS⁺03], and were the first to mention an attack against AES (without giving further details).

The rest of this section analyzes separately the cache-timing vulnerability of three components of AES-GCM: encryption, key expansion, and authentication.

3.2.1 Attacks against AES encryption

An implementation of AES which uses the precomputed lookup tables as described in Subsection 3.1.2 opens up an opportunity for a cache-timing attack. Consider, for example, the first round of AES: the indices of the table lookups are then defined simply by the xor of the plaintext and the round key K_0 . As the attacker knows or even controls the plaintext, information about the lookup indices directly leaks information about the key.

Bernstein [Ber04a] was the first to implement a cache-timing key-recovery attack against AES. While his attack relies on the attacker's capability of producing reference timing distributions from known-key encryptions on a platform identical to the target platform and has thus been deemed difficult to mount [OST06, BM06], several improved attack strategies have subsequently been described by Bertoni et al. [BZB⁺05], Osvik et al. [OST06], see also [TOS10], Aciçmez et al. [ASK06], Bonneau and Mironov [BM06], and Neve et al. [NSW06, NS07].

In particular, Osvik et al. [OST06] propose an attack model where the attacker obtains information about cache access patterns by manipulating the cache between encryptions via user-level processes. This was independent of [Ber04a] although publicized slightly later and contains the description of a fully implemented fast key recovery from Linux disk encryption. Bonneau and Mironov [BM06] further demonstrate an attack detecting cache hits in the encryption algorithm itself, as opposed to timing a process controlled by the attacker. Their attack requires no active cache manipulation, only that the tables are (partially) evicted from cache prior to the encryption. Finally, Aciçmez et al. [ASK06] note that if the encrypting machine is running multiple processes, workload on the target machine achieves the desired cache-cleaning effect, and provide simulation results suggesting that it is possible to recover an AES encryption key via a passive remote timing attack.

3.2.2 Attacks against AES key expansion

The expansion of the 128-bit AES key into 11 round keys makes use of the SUBBYTES operation which is also used for AES encryption and usually implemented through

lookup tables. During key expansion, the lookup indices are dependent on the secret key, so, in principle, ingredients for a cache-timing attack are available also during key expansion.

Mounting a cache-timing attack against AES key expansion will be very hard in practice. Common implementations do the key expansion just once and store either the fully expanded 11 round keys or partially expanded keys (see “Round-key recomputation” in Subsection 3.3.1). In both cases, table lookups based on secret data are performed just once, precluding statistical timing attacks, which require multiple timing samples.

3.2.3 Attacks against Galois/Counter Mode authentication

The computationally expensive operations for GCM authentication are multiplications in the finite field $\mathbb{F}_{2^{128}}$. More specifically, each block of input requires multiplication with a secret constant factor H derived from the master encryption key. As most common general-purpose CPUs lack support for multiplication of polynomials over \mathbb{F}_2 , the standard way of implementing GCM is through lookup tables containing precomputed multiples of H .

The specification of GCM describes different multiplication algorithms involving tables of different sizes allowing to trade memory for computation speed [MV04]. The basic idea of all of these algorithms is the same: split the non-constant factor of the multiplication into bytes or half-bytes and use these as indices for table lookups.

For the first block of input m_0 , this non-constant factor is c_0 , the first block of ciphertext. Assuming the ciphertext is available to the attacker anyway, the indices of the first block lookups do not leak any secret information. However, for the second ciphertext block c_1 , the non-constant input to the multiplication is $(c_0 \cdot H) \oplus c_1$. An attacker gaining information about this value can easily deduce the secret value H necessary for a forgery attack. Note that this does not pose any threat to the master encryption key as the authentication key H is derived from the master key via encrypting a known constant. It does give the attacker a valid plaintext-ciphertext pair.

The lookup tables used for GCM are usually at least as large as AES lookup tables; common sizes include 4 KB, 8 KB and 64 KB. The values retrieved from these tables are 16 bytes long; knowledge of the cache line of length, for example, 64 bytes (as on the Intel Core 2 processors) thus leaves only 4 possibilities for each lookup index. For example, the 64-KB implementation uses 16 tables, each corresponding to a different byte of the 128-bit input. Provided that cache hits leak the maximum 6 bits in each byte, a 2^{32} exhaustive search over the remaining unknown bits is sufficient to recover the authentication key. Common implementations of GCM are thus potentially vulnerable to authentication-key recovery via cache timing attacks.

3.3 Table-based implementations

The Rijndael algorithm was designed with a table-based implementation for 32-bit and 64-bit processors in mind (see Subsection 3.1.2); for many architectures and application scenarios this approach is still the most efficient one. This section considers “traditional”, table-based implementations of AES in counter (CTR) mode. Subsection 3.3.1 first surveys several methods to reduce the number of *instructions* required to compute AES; Subsection 3.3.2 then describes how to reduce *cycles* required to compute AES on different microarchitectures, specifically, the Motorola PowerPC G4 7410, the Intel Pentium 4 f12, the Sun UltraSPARC III, the Intel Core 2 Quad Q6600, and the AMD Athlon 64 X2 3800+.

All implementations described in this section are vulnerable to cache-timing attacks and should be used only for applications where an attacker is unable to obtain cache-timing information.

3.3.1 Reducing instructions for table-based AES

The table-based implementation of one round of AES can be decomposed into 16 shift instructions, 16 mask instructions, 16 load instructions for the table lookups, 4 load instructions for the round keys, and 16 xor instructions. Listing 1 shows C code of one round of AES decomposed into these instructions; all subsequent code samples in this section express CPU instructions using the qasm programming language (see Section 2.8); each line corresponds to one CPU instruction.

All of the target platforms have shift instructions, mask instructions, load instructions, and xor instructions. Some platforms, for example, the x86 and AMD64 architectures, do not support three-operand shift instructions (i.e., shift instructions where the output register is not the input register) but do have byte-extraction instructions that are adequate to achieve the same instruction counts. This section ignores register-allocation issues.

The 10-round main loop uses more than 680 instructions, for four reasons:

- Before the first round there are 4 extra round-key loads and 4 extra xors.
- The last round has 16 extra masks, one after each of the table lookups.
- For AES-CTR there are 4 loads of 4-byte plaintext words, 4 xors of keystream with plaintext, and 4 stores of ciphertext words.
- There are 4 extra instructions for miscellaneous tasks such as incrementing the AES-CTR input.

Overall there are 720 instructions, specifically 208 loads, 4 stores, and 508 integer instructions. The 508 integer instructions consist of 160 shift instructions, 176 mask instructions, 168 xor instructions, and 4 extra instructions.

This count ignores the costs of conditional branches; these costs are easily reduced by unrolling. It also ignores extra instructions needed to handle, e.g., big-endian loads on a little-endian architecture; almost all endianness issues can be eliminated by appropriate swapping of the AES code and tables. Furthermore it ignores

Listing 1 Round i of AES encryption, in C, input in 32-bit words y_0, y_1, y_2, y_3 , output in 32-bit words z_0, z_1, z_2, z_3

<pre> z0 = roundkeys[i * 4 + 0]; z1 = roundkeys[i * 4 + 1]; z2 = roundkeys[i * 4 + 2]; z3 = roundkeys[i * 4 + 3]; p00 = (uint32) y0 >> 20; p01 = (uint32) y0 >> 12; p02 = (uint32) y0 >> 4; p03 = (uint32) y0 << 4; p00 &= 0xff0; p01 &= 0xff0; p02 &= 0xff0; p03 &= 0xff0; p00 = *(uint32 *) (table0 + p00); p01 = *(uint32 *) (table1 + p01); p02 = *(uint32 *) (table2 + p02); p03 = *(uint32 *) (table3 + p03); z0 ^= p00; z3 ^= p01; z2 ^= p02; z1 ^= p03; p10 = (uint32) y1 >> 20; p11 = (uint32) y1 >> 12; p12 = (uint32) y1 >> 4; p13 = (uint32) y1 << 4; p10 &= 0xff0; p11 &= 0xff0; p12 &= 0xff0; p13 &= 0xff0; p10 = *(uint32 *) (table0 + p10); p11 = *(uint32 *) (table1 + p11); p12 = *(uint32 *) (table2 + p12); p13 = *(uint32 *) (table3 + p13); z1 ^= p10; z0 ^= p11; </pre>	<pre> z3 ^= p12; z2 ^= p13; p20 = (uint32) y2 >> 20; p21 = (uint32) y2 >> 12; p22 = (uint32) y2 >> 4; p23 = (uint32) y2 << 4; p20 &= 0xff0; p21 &= 0xff0; p22 &= 0xff0; p23 &= 0xff0; p20 = *(uint32 *) (table0 + p20); p21 = *(uint32 *) (table1 + p21); p22 = *(uint32 *) (table2 + p22); p23 = *(uint32 *) (table3 + p23); z2 ^= p20; z1 ^= p21; z0 ^= p22; z3 ^= p23; p30 = (uint32) y2 >> 20; p31 = (uint32) y2 >> 12; p32 = (uint32) y2 >> 4; p33 = (uint32) y2 << 4; p30 &= 0xff0; p31 &= 0xff0; p32 &= 0xff0; p33 &= 0xff0; p30 = *(uint32 *) (table0 + p30); p31 = *(uint32 *) (table1 + p31); p32 = *(uint32 *) (table2 + p32); p33 = *(uint32 *) (table3 + p33); z3 ^= p30; z2 ^= p31; z1 ^= p32; z0 ^= p33; </pre>
--	--

the initial costs of computing the 176 bytes of round keys from a 16-byte key. This computation involves hundreds of extra instructions—certainly a noticeable cost—but the round keys can be reused for all the blocks of a message. Round keys can also be reused for other messages if they are saved.

The number of instruction required for AES encryption in counter mode can be reduced using architecture-dependent techniques described in the following:

Combined shift-and-mask instructions (saving 160 instructions). Some architectures allow a shift instruction

```
p02 = y0 >> 4
```

and a mask instruction

```
p02 &= 0xff0
```

to be combined into a single instruction

```
p02 = (y0>>4) & 0xff0
```

Replacing 160 shifts and 160 masks by 160 shift-and-mask instructions saves 160 instructions.

On the 32-bit PowerPC architecture, for example, the `rlwinm` instruction can do any rotate-and-mask where the mask consists of consecutive bits.

Scaled-index loads (saving 80 instructions). On other architectures a shift instruction

```
p03 <<= 4
```

and a load instruction

```
p03 = *(uint32 *) (table3 + p03)
```

can be combined into a single instruction. The instructions

```
p03 = (uint32) y0 << 4
p03 &= 0xff0
p03 = *(uint32 *) (table3 + p03)
```

for handling the bottom byte of `y0` can then be replaced by

```
p03 = y0 & 0xff
p03 = *(uint32 *) (table3 + (p03 << 4))
```

Similarly, the instructions

```
p00 = (uint32) y0 >> 20
p00 &= 0xff0
p00 = *(uint32 *) (table0 + p00)
```

for handling the top byte of `y0` can be replaced by

```
p00 = (uint32) y0 >> 24
p00 = *(uint32 *) (table0 + (p00 << 4))
```

In 10 rounds there are 40 top bytes and 40 bottom bytes.

The x86 architecture, for example, allows scaled indices in load instructions. The x86 scaling allows only a 3-bit shift, not a 4-bit shift, but this is easily accommodated by non-interleaved (or partially interleaved) tables.

Second-byte instructions (saving 40 instructions). All architectures support a mask instruction

```
p03=y0&0xff
```

that extracts the bottom byte of `y0`.

Some platforms, for example, the x86 architecture, also support a single instruction

```
p02 = (y0 >> 8) & 0xff
```

to extract the *second* byte of *y0*. In conjunction with scaled-index loads this instruction allows

```
p02 = (uint32) y0 >> 6
p02 &= 0x3fc
p02 = *(uint32 *) (table2 + p02)
```

to be replaced by

```
p02 = (y0 >> 8) & 0xff
p02 = *(uint32 *) (table2 + (p02 << 2))
```

saving another 40 instructions overall.

Padded registers (saving 80 instructions). Some platforms, for example, the UltraSPARC architecture, do not have any of the combined instructions described above, but *do* have 64-bit registers. On these architectures one can expand a 4-byte value such as `0xc66363a5` into an 8-byte value such as `0x0c60063006300a50` (or various other possibilities such as `0x0000c60630630a50`). If this expansion is applied consistently in the registers, the lookup tables (before the last round), and the round keys, then it does not cost any extra instructions.

The advantage of the padded 8-byte value `0x0c60063006300a50` is that a single mask instruction produces the shifted bottom byte `a50`, and a single shift instruction produces the shifted top byte `c60`. Consequently the original eight shift-and-mask instructions, for extracting four shifted bytes from *y0*, can be replaced by six instructions:

```
p00 = (uint64) y0 >> 48
p01 = (uint64) y0 >> 32
p02 = (uint64) y0 >> 16
p01 &= 0xff0
p02 &= 0xff0
p03 = y0 & 0xff0
```

Expanded lookup tables, like scaled-index loads, thus save 80 instructions overall.

32-bit shifts of padded registers (saving 40 instructions). Some platforms, for example, the UltraSPARC architecture, have not only a 64-bit right-shift instruction but also a 32-bit right-shift instruction that automatically masks its 64-bit input with `0xffffffff`. This instruction, in conjunction with padded registers, allows

```
p02 = (uint64) y0 >> 16
p02 &= 0xff0
```

to be replaced by

```
p02 = (uint32) y0 >> 16
```

saving 40 additional instructions.

Byte loads (saving 4 instructions). As mentioned earlier, the last round of AES has 16 extra masks for its 16 table lookups. Four of the masks are 0xff. All of the target architectures allow these masks to be absorbed into single-byte load instructions. For example,

```
p00 = *(uint32 *) (table0 + p00)
p00 &= 0xff
```

can be replaced with

```
p00 = *(uint8*) (table0 + p00)
```

on little-endian CPUs or

```
p00 = *(uint8*) (table0 + p00 + 3)
```

on big-endian CPUs.

Two-byte loads (saving 4 instructions). Four of the masks used in the last round are 0xff00. These masks can be absorbed into two-byte load instructions if the table structure has 0x00 next to the desired byte. Often this structure occurs naturally as part of other table optimizations, and in any case it can be achieved by a separate table.

Masked tables (saving 8 instructions). The other eight masks in the last round are 0xff0000 and 0xff000000. These masked values cannot be produced by byte loads and two-byte loads but can be produced by four-byte loads from separate tables whose entries are already masked.

Separate masked tables are also the easiest way to handle the distinction between padded 64-bit registers and packed 32-bit AES output words.

Combined mask and insert (saving 16 instructions). A 4-byte result of the last round, such as z0, is produced by 4 xors with 4 masked table entries, where the masks are 0xff, 0xff00, 0xff0000, 0xff000000.

Some architectures have an instruction that replaces specified bits of one register with the corresponding bits of another register. For example, the 32-bit PowerPC architecture has a rlwimi instruction that does this, optionally rotating the second register. The instruction sequence

```
p00 &= 0xff000000
p11 &= 0xff0000
p22 &= 0xff00
p33 &= 0xff
z0 ^= p00
z0 ^= p11
z0 ^= p22
z0 ^= p33
```

can then be replaced by

```
p00 bits 0xff0000 = p11 <<< 0
p00 bits  0xff00 = p22 <<< 0
p00 bits   0xff = p33 <<< 0
z0 ^= p00
```

Note for C programmers: in C notation,

```
p00 bits 0xff0000 = p11
```

would be

```
p00 = (p00&0xff00fff)|(p11&0xff0000);
```

This is another way—without using byte loads, and without constraining the table structure—to eliminate all the extra masks.

Combined load-xor (saving 168 instructions). Often the result of a load is used solely for xoring into another register. Some architectures, for example, x86 and AMD64, allow load and xor to be combined into a single instruction.

Byte extraction via loads (saving 160 to 320 integer instructions at the cost of 200 load/store instructions). Extracting four indices from y_0 takes at most 8 integer instructions, and on some architectures as few as 4 integer instructions, as discussed above.

A completely different way to extract four bytes from y_0 —and therefore to extract indices, on architectures allowing scaled-index loads—is to store y_0 and then do four byte loads from the stored bytes of y_0 . This eliminates between 4 and 8 integer instructions—potentially helpful on CPUs where integer instructions are the main bottleneck—at the expense of 5 load/store instructions.

One can apply this conversion to all 160 byte extractions. One can also apply it to some of the byte extractions, changing the balance between load instructions and integer instructions. The optimal combination is CPU-dependent.

Round-key recomputation (saving 30 load instructions at the cost of 30 integer instructions). In the opposite direction: Instead of loading the 44 round-key words W_0, \dots, W_{43} , one can load 14 round-key words, specifically words $W_0, W_1, W_2, W_3, W_4, W_8, W_{12}, W_{16}, W_{20}, W_{24}, W_{28}, W_{32}, W_{36}, W_{40}$, and compute the other round-key words by 30 xors, taking advantage of the AES round-key structure. This reduces the number of load instructions—potentially helpful on CPUs where loads are the main bottleneck—although it increases the number of integer instructions.

One can also use intermediate combinations, such as 22 loads and 22 xors. As before, the optimal combination is CPU-dependent.

Round-key caching (saving about 44 instructions). Each 16-byte block of input involves 44 round-key loads (or xors). The same round keys are used in the next block. On an architecture with many registers, some or all of the round keys can be kept in registers, moving these loads out of the main loop. The same type of savings appears if several blocks are handled in parallel.

Counter-mode caching (saving about 100 instructions). Counter-mode encryption means generating a key stream by applying AES encryption to a 16-byte counter that is increased by 1 for every block of input. An eSTREAM AES implementation by Wu demonstrates how this specific way of applying AES can be used to save instructions:

Observe that 15 bytes of the counter remain constant for 256 blocks; just one byte of the counter changes for every block of input. All operations in the first round not depending on this byte are shared between 256 blocks. The resulting values y_0 , y_1 , y_2 and y_3 can be saved and reused in 256 consecutive blocks.

Similar observations hold for the second round: only one of the four 4-byte input words of round 2 changes every block. All computations not depending on this word can be saved and reused in 256 consecutive blocks.

3.3.2 Reducing cycles for table-based AES

Minimizing instructions is not the same as minimizing cycles. This section shows how to apply the techniques to reduce instructions on different microarchitectures and describes what other measures are necessary to address (micro-)architectural bottlenecks. This section compares the performance of the implementation on long streams of input (4096 bytes) with lower bounds derived from microarchitectural bottlenecks. For detailed benchmarking results for different input lengths see Section 3.6.

Motorola PowerPC G4 7410, 32-bit PowerPC architecture. For this CPU, the main loop consists of 461 instructions, specifically 180 load/store instructions, 279 integer instructions, and 2 branch instructions. The implementation uses the following techniques from Subsection 3.3.1:

- combined shift-and-mask instructions,
- combined mask-and-insert, and
- counter-mode caching;

it takes 14.56 cycles per byte when encrypting 4096 bytes of input (see Table 3.3).

The PowerPC G4 7410 can dispatch at most 3 instructions per cycle. At most 2 of the instructions can be load/store or integer instructions, so the 459 non-branch instructions take at least $459/2 = 229.5$ cycles, i.e., 14.34 cycles per byte. At most 1 of the instructions can be a load/store instruction, but there are only 180 load/store instructions, so this is a less important bottleneck.

The G4 is, for most purposes, an in-order CPU, so load instructions have to be interleaved with arithmetic instructions. Results of load instructions are available after 3 cycles. Saving all possible caller registers makes 29 4-byte integer registers available for AES encryption. Detailed analysis shows that these are enough registers for almost perfect instruction scheduling, making the processor execute 2 instructions almost every cycle in the AES main loop. The 14.56 cycles per byte reported by the eSTREAM benchmarking suite correspond to $14.56 \cdot 16 \approx 233$ cycles per block. This is very close to the 229.5 cycles per block lower bound for 459 instructions.

Intel Pentium 4 f12, x86 architecture. For this CPU, the main loop consists of 414 instructions. The implementation uses the following techniques from Subsection 3.3.1:

- scaled-index loads,
- second-byte instructions,
- byte loads,
- two-byte loads,
- masked tables,
- combined load-xor, and
- counter-mode caching;

it takes 14.15 cycles per byte when encrypting 4096 bytes of input (see Table 3.4).

Extra stores and loads are required to handle the extremely limited number of general-purpose x86 integer registers. The total table size (including masked tables for the last round) is compressed to 4096 bytes; this improvement does not affect the eSTREAM benchmark results but reduces cache-miss costs in many applications.

There are several tricky performance bottlenecks on the Pentium 4 [Fog10c, Fog10b, Fog10a]. The most obvious bottleneck is that the Pentium 4 can do only one load per cycle. The 14.15 cycles per byte reported by the eSTREAM benchmarking suite correspond to $14.15 \cdot 16 \approx 226$ cycles. The main loop has 177 loads, accounting for most—although certainly not all—of these 226 cycles.

Sun UltraSPARC III, UltraSPARC architecture. For this CPU, the main loop consists of 505 instructions, specifically 178 load/store instructions, 325 integer instructions, and 2 branch instructions. The implementation uses the following techniques from Subsection 3.3.1:

- padded registers;
- 32-bit shifts of padded registers;
- masked tables; and
- counter-mode caching;

it takes 12.02 cycles per byte when encrypting 4096 bytes of input (see Table 3.6).

An UltraSPARC CPU dispatches at most four instructions per cycle. Only one of these instructions can be a load/store instruction, so the 178 load/store instructions use at least 178 cycles. Furthermore, only two of these instructions can be integer instructions, so the 325 integer instructions use at least 162.5 cycles.

The simplest way to mask a byte is with an arithmetic instruction: for example, `&0xff00`. The UltraSPARC architecture supports only 12-bit immediate masks, so three of the masks have to be kept in registers.

The UltraSPARC is an in-order CPU, except for store instructions. Proper instruction scheduling thus requires each load instruction to be grouped with two integer instructions. Only 24 8-byte integer registers are available, posing some challenges for instruction scheduling. The 12.02 cycles per byte reported by the eSTREAM benchmarking suite correspond to $12.03 \cdot 16 \approx 193$ cycles per block. Register-allocation and instruction-scheduling issues are the reason why the implementation does not reach the lower bound of 178 cycles per block.

Round-key recomputation (see Subsection 3.3.1) could be used to trade some loads for integer instructions, but this makes scheduling even more difficult.

Intel Core 2 Quad Q6600 6fb, AMD64 architecture. For this CPU, the main loop consists of 434 instructions. The implementation uses the following techniques from Subsection 3.3.1:

- scaled-index loads;
- second-byte instructions;
- byte loads;
- two-byte loads;
- masked tables;
- combined load-xor;
- round-key recomputation;
- round-key caching; and
- counter-mode caching;

it takes 10.57 cycles per byte when encrypting 4096 bytes of input (see Table 3.7).

The Core 2 can dispatch three integer instructions per cycle but, like the Pentium 4, can dispatch only one load per cycle. Extra integer instructions are often spent to avoid loads and to improve the scheduling of loads. For example, the implementation keeps round-key words in “XMM” registers, even though copying an XMM register to a normal integer register costs an extra integer instruction. The 10.57 cycles per byte reported by the eSTREAM benchmarking suite correspond to $10.57 \cdot 16 \approx 169$ cycles per block. The main loop has 143 loads, accounting for most of these 169 cycles.

AMD Athlon 64 X2 3800+ 15/75/2, AMD64 architecture. For this CPU, the main loop consists of 409 instructions. The implementation uses the following techniques from Subsection 3.3.1:

- scaled-index loads;
- second-byte instructions;
- byte loads;

- two-byte loads;
- masked tables;
- combined load-xor; and
- counter-mode caching;

it takes 10.43 cycles per byte when encrypting 4096 bytes of input (see Table 3.10).

The Athlon 64 can dispatch three instructions per cycle, including *two* load instructions. Loads and stores must be carried out in program order, placing a high priority on careful instruction scheduling.

The Athlon-64-tuned software runs at 11.54 cycles per byte on the Core 2 processor, and the Core-2-tuned software runs at 14.77 cycles per byte on the Athlon 64 processor, illustrating the importance of microarchitectural differences.

3.4 Bitsliced AES for 64-bit Intel processors

The idea of bitslicing as a technique to efficiently implement cryptographic algorithms was introduced by Biham in [Bih97] for implementing DES. The idea is to hold all bits involved in a computation in different registers and simulate a hardware implementation on these registers using bit-logical instructions.

For just one computation this is usually very inefficient but on architectures with n -bit-wide registers a bitsliced implementation can perform the same computations on n different inputs in parallel which can be very fast if bit-logical operations are fast, the register width is high and of course if enough independent computations can be processed in parallel. Furthermore such implementations run in constant time and are therefore inherently secure against timing attacks.

The SSE and SSE2 instruction set extensions which are part of the AMD64 architecture support bit-logical instructions on 16 128-bit-wide XMM registers. The Intel Core 2 and Core i7 processors can dispatch up to 3 of these instructions per cycle, so these processors seem well suited for bitsliced implementations.

In the context of AES this means that a straightforward bitsliced implementation needs to work on 128 16-byte blocks in parallel; indeed this approach was used by Matsui and Nakajima to set a new speed record of 9.2 cycles per byte for AES encryption using the 128-bit XMM registers of the Intel Core 2 processor [MN07].

The big disadvantage of this implementation is that it requires 128 independent input blocks, i.e. 2 KB of input, to be processed at the same time. For applications such as hard-disk encryption, which typically have a large amount of data to encrypt, this is not a problem. When it comes to encryption of relatively short Internet packets the overhead of padding each packet to 2 KB can be huge.

The bitsliced implementation of AES described in the following follows a different approach: The main idea is that the computationally most expensive operation in AES encryption is the `SUBBYTES` operation which operates on the bytes of the AES state independently. Implementing this operation in a bitsliced way in the 128-bit XMM registers only requires 8 independent input blocks; the overhead for

row 0									row 1	row 2	row 3
column 0				column 3			
block 0	block 1	...	block 7	block 0	block 1	...	block 7

Figure 3.1: Bit ordering in one 128-bit vector of the bitsliced state

short-packet encryption is much lower. A similar approach was used by Könighofer in [Kön08], using 64-bit registers for the computation and therefore only requiring 4 independent input blocks but achieving an encryption speed of only 19.8 cycles per byte on an AMD Opteron 146 processor.

The linear layer of AES (the `SHIFTRROWS` and the `MIXCOLUMNS` operation) does not operate on the bytes of the state independently but also those operations can be implemented efficiently on 64-bit Intel processors using the `psshufb` instruction of the `SSSE3` instruction set, which is implemented on all AMD64 Intel processors.

3.4.1 Bitsliced representation of the AES state

The key to a fast bitsliced implementation is finding an efficient bitsliced representation of the cipher state. Denote the bitsliced AES state by $b[0], \dots, b[7]$, where each $b[i]$ is a 128-bit vector fitting in one XMM register. The 8 16-byte AES blocks are “sliced” bitwise, with the least significant bits of each byte in $b[0]$ and the most significant bits in the corresponding positions of $b[7]$. Now, the AES S-Box can be implemented equally efficiently whatever the order of bits within the bitsliced state. The efficiency of the linear layer, on the other hand, depends crucially on this order.

Collecting in each byte of the bitsliced state 8 bits from identical positions of 8 different AES blocks assures that bits within each byte are independent and all instructions can be kept byte-level. Furthermore, in order to simplify the `MIXCOLUMNS` operation, the 16 bytes of an AES state are collected in the state row by row. Figure 3.1 illustrates the bit ordering in each 128-bit state vector $b[i]$.

Several algorithms are known for converting the data to a bitsliced format and back [MN07, Kön08]. The implementation presented here uses 84 instructions to bitslice the input, and 8 byte shuffles to reorder the state row by row.

3.4.2 The `ADDRoundKEY` operation

The round keys are converted to bitsliced representation during key expansion. Each key is expanded to 8 128-bit values, and a round of `ADDRoundKEY` requires 8 xors from memory to the registers holding the bitsliced state.

3.4.3 The `SUBBYTES` operation

Decomposition of the `SUBBYTES` operation into bit-logical instructions uses well-known hardware implementation strategies: The starting point is the compact hard-

Table 3.1: Instruction count for the AES S-Box

	xor	and/or	mov	total
Hardware	82	35	–	117
Software	93	35	35	163

ware S-Box proposed by Canright [Can05], requiring 120 logic gates, and its improvements by Boyar and Peralta [BP09, BP10], which further reduce the gate count to 117. The implementation of the `SUBBYTES` operation is obtained by converting each logic gate (`xor`, `and`, `or`) in this implementation to its equivalent CPU instruction. In total the software version uses 163 instructions, an improvement by 15% compared to [MN07] which uses 199 instructions. Full details of the Boolean decomposition in hardware are given in the original paper [Can05], however, there are some differences between this hardware approach and the transformation to software:

First, the packed bit-logical instructions of the AMD64 instruction set have one source and one destination; i.e., one of the inputs is always overwritten by the result. Thus, extra move instructions are required whenever both inputs are reused. Furthermore, while the compact hardware implementation computes recurring Boolean subexpressions only once, not all intermediate values fit into the available 16 XMM registers. It would be possible to use extra load/store instructions to spill some intermediate values to the stack. Load and store instruction have a much higher latency than bit-logical instructions; the alternative way to deal with lack of registers is to recompute some values. Using this second approach the implementation fits entirely in the 16 registers and uses 128 packed bit-logical instructions and 35 register-to-register move instructions. Table 3.1 lists the instruction/gate counts for the `SUBBYTES` operation in software and hardware.

3.4.4 The `SHIFTRows` operation

Since each byte of the bitsliced state contains 8 bits from identical positions of 8 AES blocks, the rotation of matrix rows in the `SHIFTRows` operation corresponds to a byte permutation. With the bitsliced layout described above the bytes of each 128-bit vector need to be permuted according to the following pattern:

$$(b_{00}, b_{01}, b_{02}, b_{03}, b_{10}, b_{11}, b_{12}, b_{13}, b_{20}, b_{21}, b_{22}, b_{23}, b_{30}, b_{31}, b_{32}, b_{33}) \mapsto (b_{00}, b_{01}, b_{02}, b_{03}, b_{11}, b_{12}, b_{13}, b_{10}, b_{22}, b_{23}, b_{20}, b_{21}, b_{33}, b_{30}, b_{31}, b_{32}).$$

Using the dedicated SSSE3 byte shuffle instruction `pshufb`, the whole `SHIFTRows` operation can be done in 8 instructions. To improve performance these 8 instructions are interleaved with the `xor` instructions of the `ADDRoundKey` operation.

3.4.5 The MixCOLUMNS operation

Owing to the circularity of the multiplication matrix, each resulting byte r_{ij} can be calculated using an identical formula:

$$r_{ij} = X \cdot b_{ij} + (X + 1) \cdot b_{i+1,j} + b_{i+2,j} + b_{i+3,j},$$

where indices are reduced modulo 4.

Multiplication of b_{ij} by X in $\mathbb{F}_{2^8} = \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)$ corresponds to a left shift and a conditional xor with $0x1b$ whenever the most significant bit $b_{ij}[7] = 1$; addition corresponds to xor. For example, the least significant bit $r_{ij}[0]$ of each byte is obtained as

$$r_{ij}[0] = b_{ij}[7] \oplus b_{i+1,j}[7] \oplus b_{i+1,j}[0] \oplus b_{i+2,j}[0] \oplus b_{i+3,j}[0].$$

As the bitsliced state collects the bits of an AES state row by row, computing $b_{i+1,j}[0]$ from $b_{ij}[0]$ for all 128 least significant bits in parallel is equivalent to rotating $b[0]$ left by 32 bits:

$$(b_{00}, b_{01}, b_{02}, b_{03}, b_{10}, b_{11}, b_{12}, b_{13}, b_{20}, b_{21}, b_{22}, b_{23}, b_{30}, b_{31}, b_{32}, b_{33}) \mapsto \\ (b_{10}, b_{11}, b_{12}, b_{13}, b_{20}, b_{21}, b_{22}, b_{23}, b_{30}, b_{31}, b_{32}, b_{33}, b_{00}, b_{01}, b_{02}, b_{03}).$$

Similarly, computing $b_{i+2,j}$ requires rotation by 64 bits, computing $b_{i+3,j}$ rotation by 96 bits. To obtain the new bitsliced state vector $b[0]$, rewrite the above equation as

$$r[0] = rl^{32}b[0] \oplus rl^{64}b[0] \oplus rl^{96}b[0] \oplus b[7] \oplus rl^{32}b[7].$$

One rotation can be saved by observing that

$$rl^{64}b[i] \oplus rl^{96}b[i] = rl^{64}(b[i] \oplus rl^{32}b[i]),$$

yielding

$$r[0] = rl^{32}b[0] \oplus rl^{64}(b[0] \oplus rl^{32}b[0]) \oplus b[7] \oplus rl^{32}b[7].$$

Thus, only two rotations are required per register, 16 in total.

Equations for the other state vectors $r[i]$ are derived in a similar way:

$$r[1] = b[0] \oplus rl^{32}b[0] \oplus rl^{32}b[1] \oplus rl^{64}(b[1] \oplus rl^{32}b[1]) \oplus \\ b[7] \oplus rl^{32}b[7],$$

$$r[2] = b[1] \oplus rl^{32}b[1] \oplus rl^{32}b[2] \oplus rl^{64}(b[2] \oplus rl^{32}b[2]),$$

$$r[3] = b[2] \oplus rl^{32}b[2] \oplus rl^{32}b[3] \oplus rl^{64}(b[3] \oplus rl^{32}b[3]) \oplus \\ b[7] \oplus rl^{32}b[7],$$

$$r[4] = b[3] \oplus rl^{32}b[3] \oplus rl^{32}b[4] \oplus rl^{64}(b[4] \oplus rl^{32}b[4]) \oplus \\ b[7] \oplus rl^{32}b[7],$$

$$r[5] = b[4] \oplus rl^{32}b[4] \oplus rl^{32}b[5] \oplus rl^{64}(b[5] \oplus rl^{32}b[5]),$$

$$r[6] = b[5] \oplus rl^{32}b[5] \oplus rl^{32}b[6] \oplus rl^{64}(b[6] \oplus rl^{32}b[6]),$$

$$r[7] = b[6] \oplus rl^{32}b[6] \oplus rl^{32}b[7] \oplus rl^{64}(b[7] \oplus rl^{32}b[7]).$$

There is no dedicated rotate instruction for XMM registers; however, all the rotations are by multiples of 4 bytes and can be implemented using the `pshufd` 32-bit-doubleword permutation instruction. This instruction supports writing the result in a destination register different from the source register and thus saves register-to-register moves. In total, the implementation of `MIXCOLUMNS` requires 43 instructions: 16 `pshufd` instructions and 27 `xors`.

3.4.6 Key expansion

The AES key expansion algorithm computes 10 round keys from the initial key, using a sequence of `SUBBYTES` operations and `xors`. The constant-time implementation of this expansion uses the input/output transformation to and from bitsliced format, and the implementation of `SUBBYTES`. It performs 10 unavoidably sequential `SUBBYTES` operations; its cost is thus roughly equivalent to the cost of one 8-block AES encryption. The performance results in Section 3.6 include exact cycle count for this constant-time key expansion and a lookup-table-based key expansion.

3.5 AES-GCM for 64-bit Intel processors

This section describes an extension of the bitsliced AES-CTR implementation described in Section 3.4 to support authenticated encryption using AES-GCM.

As described in Subsection 3.1.4, the core operation required for GCM authentication is multiplication in $\mathbb{F}_{2^{128}}$ with a secret constant element H . This section describes two different implementations of this multiplication: first, a standard table-based approach, and second, a constant-time solution. Both implementations consist of a one-time key schedule computing H and tables containing multiples of H ; and an on-line phase which performs the actual authentication. Both implementations accept standard (non-bitsliced) input.

3.5.1 Table-based implementation

Several flavors of Galois field multiplication involving lookup tables of different sizes have been proposed for GCM software implementation [MV04]. The “simple, 4-bit tables method” uses 32 tables with 16 precomputed multiples of H each, corresponding to a memory requirement of 8 KB. Using this approach one multiplication by H is performed using 84 arithmetic instructions and 32 128-bit load, following the ideas from [Dai09b].

The computation is free of long chains of dependent instructions and the computation is thus mainly bottlenecked by the number of 32 loads per multiplication yielding a performance of 10.68 cycles per byte for full AES-GCM on a Core 2 Q9550. For detailed performance figures see Subsection 3.6.2.

Algorithm 4 Multiplication in $\mathbb{F}_{2^{128}}$ of D with a constant element H .

Input: Input D , precomputed values $H, X \cdot H, X^2 \cdot H, \dots, X^{127} \cdot H$

Output: Product DH

```

DH = 0
for i from 0 to 127 do
  if  $d_i = 1$  then
     $DH \leftarrow DH \oplus X^i \cdot H$ 
  end if
end for

```

3.5.2 Constant-time implementation

The constant-time implementation of GCM authentication does not use any table lookups indexed by secret data or data-dependent branches and is thus immune to timing attacks. While slower than the implementation described in Subsection 3.5.1, the constant-time implementation achieves a reasonable speed of 21.99 cycles per encrypted and authenticated byte on a Core 2 Q9550 and, in addition, requires only 2 KB of memory for precomputed values, comparing favorably to lookup-table-based implementations.

The offline phase precomputes values $H, X \cdot H, X^2 \cdot H, \dots, X^{127} \cdot H$. Based on this precomputation, multiplication of an element D with H is computed using a series of xors conditioned on the bits of D , as shown in Algorithm 4.

For a constant-time version of this algorithm conditional statements have to be replaced by a sequence of instructions that take deterministic time. The following sequence of 6 instructions shows the constant-time implementation of a conditional xor of register `xmm3` into register `xmm4` if and only if bit b_0 of `xmm0` is set:

```

movdqa  %xmm0, %xmm1
pand    BIT0, %xmm1          # BIT0 - bit mask in memory
pcmpeqd BIT0, %xmm1
pshufd  $0xff, %xmm1, %xmm1
pand    %xmm3, %xmm1
pxor    %xmm1, %xmm4

```

The first four instructions produce an all-zero mask in the register `xmm1` if $b_0 = 0$ and an all-one mask otherwise. The last two instructions mask `xmm3` with this value and xor the result.

The precomputation in the offline phase described above is implemented in constant time using the same conditional-xor technique. The multiplications with X in this offline phase require an additional right shift. As there is no dedicated 1-bit shift for XMM registers; it is implemented through two packed 64-bit right shifts and correction of the least significant bit of the rightmost 64-bit word using another conditional xor.

Each 128-bit multiplication in the online phase loops through all 128 bits of the intermediate value D . Each loop requires $6 \cdot 128$ instructions, or 48 instructions per

byte. The above sequence of six instructions for a conditional xor can in this context be optimized by considering four bitmasks in parallel and only repeating lines 1–3 of the code once every four bits, yielding a final complexity of 3.75 instructions per bit, or 30 instructions/byte. As the Core 2 processor can issue at most 3 arithmetic instructions per cycle, a theoretical lower bound for a single Galois field multiplication with the above implementation of the conditional xor, is 10 cycles per byte. The actual performance comes rather close at around 14 cycles per byte for the complete authentication.

3.6 Performance results and comparison

Several implementations of AES-CTR have been included in the eSTREAM benchmarking framework [Can08] and subsequently been integrated into the SUPERCOP framework [BLa]. The eSTREAM benchmarking framework, unlike SUPERCOP, reports separate cycle counts for key setup, IV setup, and for encryption. Separating these operations in the benchmarks is important especially for the bitsliced implementation which is why the eSTREAM benchmarking suite is used for the benchmarks. This benchmarking suite compiles each algorithm with all available compilers and a large list of compiler options and then reports the fastest results. The “simple Imix” reported by the eSTREAM benchmarking suite is a weighted average simulating sizes of typical IP packets: it takes into account packets of size 40 bytes (7 parts), 576 bytes (4 parts), and 1500 bytes (1 part). The computers used for benchmarking are described in Table 3.2.

Table 3.2: Machines used for benchmarking the implementations of AES and AES-GCM

gggg		
Affiliation	University of Illinois at Chicago	
CPU	PowerPC G4 7410, 533.33 MHz	
Compilers	gcc	gcc (Debian 4.4.4-17) 4.4.5
	gcc-4.3	gcc-4.3 (Debian 4.3.5-3) 4.3.5
fireball		
Affiliation	University of Illinois at Chicago	
CPU	Intel Pentium 4 f12, 1894.135 MHz	
Compilers	gcc	gcc (Ubuntu 4.4.1-4ubuntu9) 4.4.1
	gcc-4.1	gcc-4.1 (GCC) 4.1.3 20080704 (prerelease) (Ubuntu 4.1.2-27ubuntu1)
	gcc-4.2	gcc-4.2 (GCC) 4.2.4 (Ubuntu 4.2.4-5ubuntu1)
	gcc-4.3	gcc-4.3 (Ubuntu 4.3.4-5ubuntu1) 4.3.4
	g++	g++ (Ubuntu 4.4.1-4ubuntu9) 4.4.1
nmi-0039		
Affiliation	NMI Build and Test Lab at the University of Wisconsin at Madison	

CPU	Sun UltraSparc IIIi, 1062 MHz	
Compilers	gcc-3.4 gcc-4.0 gcc-4.1 gcc-4.2 gcc-4.3 cc	sparc-sun-solaris2.10-gcc-3.4.3 (GCC) 3.4.3 sparc-sun-solaris2.10-gcc-4.0.4 (GCC) 4.0.4 sparc-sun-solaris2.10-gcc-4.1.2 (GCC) 4.1.2 sparc-sun-solaris2.10-gcc-4.2.4 (GCC) 4.2.4 sparc-sun-solaris2.10-gcc-4.3.2 (GCC) 4.3.2 Forte Developer 7 C 5.4 2002/03/09
smirk		
Affiliation	Peter Schwabe	
CPU	Sun UltraSparc IIIi (Sabre), 333 MHz	
Compilers	gcc gcc-4.1 gcc-4.2	gcc (Debian 4.3.2-1.1) 4.3.2 gcc-4.1 (GCC) 4.1.3 20080420 (prerelease) (Debian 4.1.2-22) (GCC) 4.2.4 (Debian 4.2.4-2+b1)
latour		
Affiliation	Eindhoven University of Technology	
CPU	Intel Core 2 Quad Q6600, 2394 MHz	
Compilers	gcc gcc-4.1 gcc-4.2 g++	gcc (Ubuntu 4.3.3-5ubuntu4) 4.3.3 gcc-4.1 (GCC) 4.1.3 20080623 (prerelease) (Ubuntu 4.1.2-24ubuntu1) gcc-4.2 (GCC) 4.2.4 (Ubuntu 4.2.4-5ubuntu1) g++ (Ubuntu 4.3.3-5ubuntu4) 4.3.3
berlekamp		
Affiliation	National Taiwan University	
CPU	Intel Core 2 Quad Q9550, 2833 MHz	
Compilers	gcc gcc34 g++	gcc (GCC) 4.4.1 20090725 (Red Hat 4.4.1-2) gcc34 (GCC) 3.4.6 20060404 (Red Hat 3.4.6-13) g++ (GCC) 4.4.1 20090725 (Red Hat 4.4.1-2)
dragon3		
Affiliation	Academia Sinica	
CPU	Intel Xeon X5650, 2668 MHz	
Compilers	gcc g++	gcc (GCC) 4.4.4 20100630 (Red Hat 4.4.4-10) g++ (GCC) 4.4.4 20100630 (Red Hat 4.4.4-10)
mace		
Affiliation	University of Illinois at Chicago	
CPU	AMD Athlon 64 X2 3800+, 2008.818 MHz	

Compilers	gcc	gcc (Ubuntu 4.4.3-4ubuntu5) 4.4.3
	gcc-4.0	gcc-4.0 (GCC) 4.0.4 20060630 (prerelease) (Ubuntu 4.0.3-4)
	gcc-4.1	gcc-4.1 (GCC) 4.1.3 20080704 (prerelease) (Ubuntu 4.1.2-27ubuntu1)
	gcc-4.3	gcc-4.3 (Ubuntu 4.3.4-10ubuntu1) 4.3.4
	g++	g++ (Ubuntu 4.4.3-4ubuntu5) 4.4.3

3.6.1 Benchmarks of AES-CTR

Tables 3.3 to 3.10 compare the performance of the implementations described in Sections 3.3 and 3.4 of AES in counter mode with the other implementations in the eSTREAM framework. The tables also report the fastest performance claims from unpublished software described in the literature obtained on similar hardware. They furthermore report performance figures of the AES-CTR implementation of the Crypto++ library (version 5.6.1) [Dai09b] benchmarked in the SUPERCOP benchmarking framework [BLa] for machines on which the library builds without errors. The SUPERCOP benchmarking suite includes cost for key setup and IV setup in the encryption and reports extrapolated figures for “long-stream encryption”, which do not include this cost. Note that the AES-CTR implementation in the Crypto++ library is not compatible with the eSTREAM implementations; it handles the counter in a slightly different way.

Table 3.3: Cycles/byte for AES-CTR encryption on gggg and literature claims on PowerPC G4 processors

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.3 ^c	14.56	14.72	15.34	25.66	15.82
	Key setup: 201.60 cycles ^c				
Bernstein ^{a,d}	19.11	19.30	19.45	29.21	20.06
	Key setup: 165.60 cycles ^{a,e}				
Gladman ^{b,f}	27.27	27.41	27.42	35.33	27.96
	Key setup: 193.60 cycles ^{b,g}				
Literature claims					
Code by Ahrens [Lip06]: 25.06 cycles/byte on a PowerPC G4 7400					
Code by Ahrens [Lip06]: 24.06 cycles/byte on a PowerPC G4 7457					

^a Implementation `bernstein/big-1/1`

^b Implementation `gladman/1`

^c Compiled with `gcc-3.4 -mcpu=G3 -Os -fomit-frame-pointer`

^d Compiled with `gcc-3.4 -mcpu=G3 -O3 -fomit-frame-pointer`

^e Compiled with `gcc-3.4 -mcpu=G3 -O3 -unroll-all-loops -fomit-frame-pointer`

^f Compiled with `gcc-3.4 -mcpu=G3 -O2 -fomit-frame-pointer`

^g Compiled with `gcc-4.2 -mcpu=G5 -O3 -fomit-frame-pointer`

Table 3.4: Cycles/byte for AES-CTR encryption on fireball and literature claims on Intel Pentium 4 processors

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.3 ^e	14.15	14.38	14.70	29.29	15.74
	Key setup: 402.61 cycles ^d				
Bernstein ^{a,f}	16.97	17.15	17.46	27.64	18.04
	Key setup: 402.61 cycles ^d				
Wu ^{c,g}	21.33	21.68	22.38	34.27	22.94
	Key setup: 607.46 cycles ^h				
Gladman ^{b,h}	29.98	30.32	30.	38.04	30.91
	Key setup: 291.51 cycles ^e				
SUPERCOP benchmark					
[Dai09b] ⁱ	19.24	n/a	27.81	n/a	n/a
	Long stream (extrapolated): 17.92 cycles/byte ⁱ				
Literature claims					
[Osv03]: 16.25 cycles/byte on an unspecified Pentium 4					
[MF05]: 15.69 cycles/byte on a Pentium 4 Northwood					
[MF05]: 17.75 cycles/byte on a Pentium 4 Prescott					
[Mat06]: 17.75 cycles/byte on an unspecified Pentium 4 32-bit processor					
[Lip06]: 15.88 cycles/byte on an unspecified Pentium 4					

^a Implementation `bernstein/x86mmx-1/1`

^b Implementation `gladman/1`

^c Implementation `hongjun/v1/1`

^d Compiled with `gcc -march=pentium -Os -fomit-frame-pointer`

^e Compiled with `gcc -march=pentiumpro -Os -fomit-frame-pointer`

^f Compiled with `gcc -march=pentium-m -Os -fomit-frame-pointer`

^g Compiled with `gcc -march=pentium4 -O1 -fomit-frame-pointer`

^h Compiled with `gcc -march=k8 -O1 -fomit-frame-pointer`

ⁱ Compiled with `g++ -m32 -march=pentium4 -O2 -fomit-frame-pointer`

Table 3.5: Cycles/byte for AES-CTR encryption on smirk and literature claims on UltraSparc II processors

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.3 ^c	12.12	12.95	14.54	24.56	14.64
	Key setup: 847.33 cycles ^d				
Bernstein ^{a,e}	20.64	21.20	21.59	32.87	22.22
	Key setup: 172.46 cycles ^e				
Gladman ^{b,f}	27.80	28.10	28.64	39.61	29.19
	Key setup: 184.02 cycles ^g				
Literature claims					
[Lip06]: 16.875 cycles/byte on a 480 MHz Sparc, presumably an UltraSparc II					

^a Implementation `bernstein/big-1/1`^b Implementation `gladman/1`^c Compiled with `gcc-4.1 -mcpu=ultrasparc3 -m64 -Os -fomit-frame-pointer`^d Compiled with `gcc-4.2 -mcpu=ultrasparc3 -m64 -Os -fomit-frame-pointer`^e Compiled with `gcc-4.2 -mcpu=ultrasparc -m64 -O3 -funroll-all-loops -fomit-frame-pointer`^f Compiled with `gcc-4.1 -mcpu=ultrasparc -Os -fomit-frame-pointer`^g Compiled with `gcc-4.2 -mcpu=ultrasparc -O3 -fomit-frame-pointer`

Table 3.6: Cycles/byte for AES-CTR encryption on nmi-0039

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.3 ^c	12.02	12.24	12.23	20.82	12.82
	Key setup: 707.14 cycles ^d				
Bernstein ^{a,e}	19.70	19.94	20.15	33.17	20.97
	Key setup: 167.21 cycles ^f				
Gladman ^{b,g}	25.06	25.22	25.25	33.64	25.82
	Key setup: 197.21 cycles ^e				

^a Implementation `bernstein/big-1/1`^b Implementation `gladman/1`^c Compiled with `gcc-4.2 -mcpu=ultrasparc3 -m64 -Os -fomit-frame-pointer`^d Compiled with `gcc -mcpu=ultrasparc3 -m64 -Os -fomit-frame-pointer`^e Compiled with `gcc-4.2 -mcpu=ultrasparc -O3 -funroll-all-loops -fomit-frame-pointer`^f Compiled with `gcc-4.2 -mcpu=ultrasparc3 -O2 -fomit-frame-pointer`^g Compiled with `cc -xarch=v7 -fast`

Table 3.7: Cycles/byte for AES-CTR encryption on 1atour and literature claims on 65-nm Intel Core 2 processors

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.4 ^f	9.32	9.76	10.76	34.30	12.01
	Key setup (table-based): 770.53 cycles ^f Key setup (constant-time): 1406.61 cycles ^f				
Section 3.3 ^d	10.57	10.78	10.79	19.51	11.38
	Key setup: 166.08 cycles ^d				
Wu ^{c,i}	12.28	12.43	12.53	18.17	12.88
	Key setup: 266.83 cycles ^g				
Bernstein ^{a,e}	13.74	13.86	13.90	19.26	14.25
	Key setup: 166.08 cycles ^h				
Gladman ^{b,i}	16.11	16.25	16.24	20.67	16.55
	Key setup: 167.43 cycles ^j				
SUPERCOP benchmark					
[Dai09b] ^k	12.61	n/a	16.03	n/a	n/a
	Long stream (extrapolated): 12.10 cycles/byte ^k				
Literature claims					
[MN07]: 9.19 cycles/byte without transformation to and from bitsliced format on an unspecified Core 2 processor					
[MN07]: 14.5 cycles/byte using lookup tables					

^a Implementation `bernstein/amd64-2/1`

^b Implementation `gladman/1`

^c Implementation `hongjun/v1/1`

^d Compiled with `gcc-4.1 -Os -fomit-frame-pointer`

^e Compiled with `gcc-4.1 -march=nocona -Os -fomit-frame-pointer`

^f Compiled with `gcc-4.2 -Os -fomit-frame-pointer`

^g Compiled with `gcc-4.2 -O3 -funroll-all-loops -fomit-frame-pointer`

^h Compiled with `gcc-4.2 -march=k8 -Os -fomit-frame-pointer`

ⁱ Compiled with `gcc-4.2 -march=nocona -Os -fomit-frame-pointer`

^j Compiled with `gcc-4.3 -march=prescott -m32 -O3 -fomit-frame-pointer`

^k Compiled with `g++ -O2 -fomit-frame-pointer`

Table 3.8: Cycles/byte for AES-CTR encryption on berlekamp

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.4 ^d	7.58	7.96	8.84	28.72	9.88
	Key setup (table-based): 778.48 cycles ^e				
	Key setup (constant-time): 1144.16 cycles ^e				
Section 3.3 ^e	10.58	10.76	10.76	19.69	11.37
	Key setup: 168.01 cycles ^g				
Wu ^{c,f}	12.30	12.45	12.53	17.86	12.87
	Key setup: 385.41 cycles ^g				
Bernstein ^{a,g}	13.75	13.86	13.94	19.57	14.30
	Key setup: 168.01 cycles ^g				
Gladman ^{b,g}	16.22	16.40	16.35	20.77	16.67
	Key setup: 166.41 cycles ^g				
SUPERCOP benchmark					
[Dai09b] ^h	12.58	n/a	15.66	n/a	n/a
	Long stream (extrapolated): 12.11 cycles/byte ^h				

^a Implementation `bernstein/amd64-2/1`^b Implementation `gladman/1`^c Implementation `hongjun/v1/1`^d Compiled with `gcc -Os -fomit-frame-pointer`^e Compiled with `gcc -march=nocona -Os -fomit-frame-pointer`^f Compiled with `gcc -march=nocona -O2 -fomit-frame-pointer`^g Compiled with `gcc -march=k8 -Os -fomit-frame-pointer`^h Compiled with `g++ -m64 -march=nocona -O2 -fomit-frame-pointer`

Table 3.9: Cycles/byte for AES-CTR encryption on dragon3

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.4 ^d	6.68	7.04	7.80	25.47	8.73
	Key setup (table-based): 722.13 cycles ^e				
	Key setup (constant-time): 993.63 cycles ^e				
Section 3.3 ^d	9.64	9.77	9.74	17.58	10.29
	Key setup: 153.14 cycles ^d				
Wu ^{c,f}	10.84	10.85	10.92	15.44	11.20
	Key setup: 374.28 cycles ^e				
Bernstein ^{a,e}	12.52	12.62	12.68	17.32	12.98
	Key setup: 153.31 cycles ^d				
Gladman ^{b,e}	13.66	13.65	13.66	17.39	13.91
	Key setup: 156.71 cycles ^e				
SUPERCOP benchmark					
[Dai09b] ^g	11.11	n/a	14.22	n/a	n/a
	Long stream (extrapolated): 10.62 cycles/byte ^g				

^a Implementation `bernstein/amd64-1/1`

^b Implementation `gladman/1`

^c Implementation `hongjun/v1/1`

^d Compiled with `gcc -Os -fomit-frame-pointer`

^e Compiled with `gcc -march=nocona -Os -fomit-frame-pointer`

^f Compiled with `gcc -march=nocona -O2 -fomit-frame-pointer`

^g Compiled with `g++ -m64 -march=core2 -O3 -fomit-frame-pointer`

Table 3.10: Cycles/byte for AES-CTR encryption on mace and literature claims on 64-bit AMD processors

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.3 ^f	10.43	10.57	10.70	16.88	11.07
	Key setup: 191.21 cycles ^d				
Wu ^{c,h}	13.32	13.45	13.58	19.89	13.96
	Key setup: 356.92 cycles ^f				
Bernstein ^{a,e}	13.40	13.38	13.46	19.18	13.82
	Key setup: 191.21 cycles ^e				
Gladman ^{b,g}	18.28	18.27	18.32	23.78	18.68
	Key setup: 198.01 cycles ^h				
SUPERCOP benchmark					
[Dai09b] ⁱ	12.66	n/a	16.12	n/a	n/a
	Long stream (extrapolated): 12.15 cycles/byte ^t				
Literature claims					
[Mat06]: 10.63 cycles/byte on an unspecified Athlon 64 processor					
[Kön08]: 19.81 cycles/byte on an AMD Opteron 146 processor					

^a Implementation `bernstein/amd64-1/1`

^b Implementation `gladman/1`

^c Implementation `hongjun/v1/1`

^d Compiled with `gcc-4.0 -Os -fomit-frame-pointer`

^e Compiled with `gcc-4.1 -Os -fomit-frame-pointer`

^f Compiled with `gcc-4.1 -march=nocona -Os -fomit-frame-pointer`

^g Compiled with `gcc-4.1 -march=k8 -O3 -funroll-all-loops -fomit-frame-pointer`

^h Compiled with `gcc-4.4 -march=prescott -m32 -O3 -funroll-all-loops -fomit-frame-pointer`

ⁱ Compiled with `g++ -m64 -march=nocona -O3 -fomit-frame-pointer`

Table 3.11: Cycles/byte for AES-GCM encryption and authentication on latour and literature claims, numbers in paratheses are accumulated numbers including cost for IV setup and finalization

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.5.1 ^a	12.21 (12.54)	13.71 (14.62)	16.21 (18.58)	77.14 (111.28)	19.47 (23.48)
Section 3.5.2 ^c	27.14 (27.53)	28.82 (29.89)	31.53 (34.30)	100.10 (140.04)	35.24 (39.93)
Precomputation and key setup, table-based: 2917.76 cycles ^a					
Precomputation and key setup, constant-time: 4234.64 cycles ^d					
IV setup and finalization, table-based: 1365.41 cycles ^b					
IV setup and finalization, constant-time: 1597.77 cycles ^a					
Literature claims					
[Dai09a] 17.2 cycles/byte for authenticated encryption using 2-KB tables, [Dai09a] 16.1 cycles/byte for authenticated encryption using 64-KB tables, both on an unspecified Intel Core 2 processor running in 32-bit mode					

^a Compiled with gcc-4.1 -Os -fomit-frame-pointer

^b Compiled with gcc-4.2 -march=k8 -Os -fomit-frame-pointer

^c Compiled with gcc-4.2 -Os -fomit-frame-pointer

^d Compiled with gcc-4.1 -march=k8 -Os -fomit-frame-pointer

3.6.2 Benchmarks of AES-GCM

For AES-GCM authenticated encryption, the eSTREAM benchmarking suite reports cycles per encrypted and authenticated byte without considering finalization (one 16-byte AES encryption and one multiplication) necessary to compute the authentication tag. Cycles required for these final computations are reported as part of the IV setup. Tables 3.11, 3.12 and 3.13 therefore give performance numbers as reported by the eSTREAM benchmarking suite (cycles per byte and cycles required for IV setup) and “accumulated” cycles per byte, illustrating the actual time required for authenticated encryption.

For AES-GCM, there exist no benchmarking results from open benchmarking suites such as the eSTREAM suite or SUPERCOP [BLa]. The designers of GCM provide performance figures for 128-bit AES-GCM measured on a Motorola G4 processor which is certainly not comparable to an Intel Core 2 [MV05]. Comparison with the benchmarking results of the Crypto++ library by Dai on an unspecified Intel Core 2 processor running in 32-bit mode [Dai09a] is given in Table 3.11.

Table 3.12: Cycles/byte for AES-GCM encryption authentication on `berlekamp`, numbers in paratheses are accumulated numbers including cost for IV setup and finalization

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.5.1 ^a	10.40 (10.68)	11.67 (12.43)	13.82 (15.79)	65.87 (94.20)	16.60 (19.93)
Section 3.5.2 ^c	21.67 (21.99)	23.04 (23.91)	25.37 (27.63)	82.99 (115.58)	28.46 (32.29)
Precomputation and key setup, table-based: 2633.53 cycles ^b					
Precomputation and key setup, constant-time: 3454.27 cycles ^c					
IV setup and finalization, table-based: 1133.32 cycles ^c					
IV setup and finalization, constant-time: 1303.54 cycles ^b					

^a Compiled with `gcc -march=k8 -Os -fomit-frame-pointer`

^b Compiled with `gcc -Os -fomit-frame-pointer`

^c Compiled with `gcc -march=nocona -Os -fomit-frame-pointer`

Table 3.13: Cycles/byte for AES-GCM encryption and authentication on `dragon3`, numbers in paratheses are accumulated numbers including cost for IV setup and finalization

Packet size	4096 bytes	1500 bytes	576 bytes	40 bytes	Imix
Section 3.5.1 ^a	9.28 (9.53)	10.36 (11.04)	12.23 (13.99)	56.23 (81.58)	14.56 (17.54)
Section 3.5.2 ^a	19.17 (19.45)	20.39 (21.15)	22.32 (24.31)	71.53 (100.15)	24.99 (28.35)
Precomputation and key setup, table-based: 2269.28 cycles ^a					
Precomputation and key setup, constant-time: 3283.80 cycles ^a					
IV setup and finalization, table-based: 1013.80 cycles ^b					
IV setup and finalization, constant-time: 1144.62 cycles ^b					

^a Compiled with `gcc -march=k8 -Os -fomit-frame-pointer`

^k Compiled with `gcc -march=nocona -Os -fomit-frame-pointer`

4

Elliptic-curve cryptography on Cell processors

Elliptic-curve cryptography, proposed independently by Koblitz in [Kob87] and Miller in [Mil86b], is now the state of the art in public-key cryptography. The reason why cryptosystems based on arithmetic in elliptic-curve groups is preferred over systems based on the multiplicative group \mathbb{F}_q^* is that the underlying number-theoretic problem, the discrete-logarithm problem, is believed to be harder for elliptic-curve groups. More specifically, in most large prime-order subgroups of elliptic-curve groups there is no algorithm known which solves this problem in subexponential time, while the discrete-logarithm problem in \mathbb{F}_q^* can be solved using an index-calculus algorithm that takes subexponential time. Elliptic-curve cryptography can thus use significantly smaller key sizes than \mathbb{F}_q^* -based cryptosystems on the same security level and is therefore much more efficient.

This chapter describes an implementation of the most fundamental cryptographic protocol based on elliptic curves: elliptic-curve Diffie-Hellman (ECDH) key exchange. The implementation targets the Synergistic Processor Units (SPUs) of the Cell Broadband Engine [CRDI05]. The Cell Broadband Engine processor contains eight of these SPUs and can for example be found in the IBM QS20 and QS21 blade servers and the Sony PlayStation 3. The Sony PlayStation 3 makes only 6 SPUs available to the programmer. From an implementor's perspective the architecture is a particularly interesting challenge because its small integer multipliers do not seem to make it the best-suited platform for public-key cryptography. This chapter shows that it is possible to achieve performance competitive to other architectures by

- parting with the traditional way of implementing elliptic-curve cryptography which uses arithmetic operations in the underlying field as smallest building blocks,

- representing finite -field elements in a way that takes into account the special structure of the finite field and the unconventional SPU instruction set, and
- carefully optimizing the code on the assembly level.

The implementation takes constant time independent of secret inputs and is thus protected against timing attacks.

Section 4.1 gives the mathematical and algorithmic background on elliptic-curve Diffie-Hellman key exchange and introduces the Curve25519 key-exchange function. Section 4.2 gives implementation details and explains the optimization techniques used to make the software run fast on the SPUs of the Cell Broadband Engine. Finally Section 4.3 reports benchmarking results from various machines.

This chapter is based on joint work with Costigan published in [CS09]. Aside from some minor changes to notation and phrasing the main differences between the content of this chapter and [CS09] are the following:

- the performance analysis of IBM's MPM library contained in [CS09] is not contained in this chapter; and
- the benchmarking results in Section 4.3 are obtained on a slightly different set of machines than in [CS09].

4.1 Elliptic-curve Diffie-Hellman key exchange and the Curve25519 function

This section gives the mathematical and algorithmic background on elliptic-curve Diffie-Hellman key exchange necessary to understand the target function and the implementation. For a comprehensive introduction to elliptic-curve cryptography (ECC) with a focus on algorithmic aspects see for example [HMOV04].

4.1.1 Elliptic-Curve Diffie-Hellman key exchange (ECDH)

Let \mathbb{F}_q be a finite field and E/\mathbb{F}_q an elliptic curve defined over \mathbb{F}_q . Let $E(\mathbb{F}_q)$ denote the group of \mathbb{F}_q -rational points on E , and $[m]P$ the m -th scalar multiple of a point $P \in E(\mathbb{F}_q)$ for any $m \in \mathbb{Z}$.

The Diffie-Hellman key exchange protocol [DH76] can now be carried out in the group $\langle P \rangle \subseteq E(\mathbb{F}_q)$ as follows:

- User A chooses a random $a \in \{2, \dots, |\langle P \rangle| - 1\}$, computes $[a]P$ and sends this to user B .
- User B chooses a random $b \in \{2, \dots, |\langle P \rangle| - 1\}$, computes $[b]P$ and sends this to user A .
- Now both users can compute $Q = [a]([b]P) = [b]([a]P) = [(a \cdot b)]P$.

The joint key for secret-key cryptography is then extracted from Q ; a common way to do this is to compute a hash value of the x -coordinate of Q .

4.1.2 Montgomery ladder for scalar multiplication

For elliptic curves defined by an equation of the form $By^2 = x^3 + Ax^2 + x$, Montgomery introduced in [Mon87] a fast method to compute the x -coordinate of a point $R = P + Q$, given the x -coordinates of two points P and Q and the x -coordinate of their difference $P - Q$. These formulas lead to an efficient algorithm to compute the x -coordinate of $Q = [k]P$ for any point P . This algorithm is often referred to as the Montgomery ladder. In this algorithm the x -coordinate x_P of a point P is represented as (X_P, Z_P) , where $x_P = X_P/Z_P$; for the representation of the point at infinity see the discussion in [Ber06, Appendix B]. See Algorithms 5 and 6 for a pseudocode description of the Montgomery ladder.

Algorithm 5 The Montgomery ladder for x -coordinate-based scalar multiplication on the elliptic curve $E : By^2 = x^3 + Ax^2 + x$

Input: An integer $t > 0$, a scalar k with $0 \leq k \leq 2^t - 1$ and the x -coordinate x_P of some point P

Output: $(X_{[k]P}, Z_{[k]P})$ fulfilling $x_{[k]P} = X_{[k]P}/Z_{[k]P}$

$X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$

for $i \leftarrow t - 1$ **downto** 0 **do**

if bit i of k is 1 **then**

$(X_3, Z_3, X_2, Z_2) \leftarrow \text{LADDERSTEP}(x_P, X_3, Z_3, X_2, Z_2)$

else

$(X_2, Z_2, X_3, Z_3) \leftarrow \text{LADDERSTEP}(x_P, X_2, Z_2, X_3, Z_3)$

end if

end for

return (X_2, Z_2)

Each “ladder step” as described in Algorithm 6 requires 5 multiplications, 4 squarings, 8 additions and one multiplication with the constant $(A+2)/4$ in the underlying finite field.

4.1.3 The Curve25519 function

Bernstein proposed the Curve25519 function for elliptic-curve Diffie-Hellman key exchange [Ber06]. This function uses arithmetic on the elliptic curve defined by the equation $E : y^2 = x^3 + Ax^2 + x$ over the field \mathbb{F}_p , where $p = 2^{255} - 19$ and $A = 486662$; observe that on this elliptic curve it is possible to use the x -coordinate-based scalar multiplication described above.

The elliptic curve and underlying finite field are carefully chosen to meet high security requirements and to enable a fast implementation. For a detailed discussion of the security properties of Curve25519 see [Ber06, Section 3].

The Curve25519 function takes as input two 32-byte strings, one representing the x -coordinate of a point P and the other representing a 256-bit scalar k . It gives as output a 32-byte string representing the x -coordinate x_Q of $Q = [k]P$. For each of these values Curve25519 is assuming little-endian representation.

Algorithm 6 One ladder step of the Montgomery ladder

```

function LADDERSTEP( $x_{Q-P}, X_P, Z_P, X_Q, Z_Q$ )
   $t_1 \leftarrow X_P + Z_P$ 
   $t_6 \leftarrow t_1^2$ 
   $t_2 \leftarrow X_P - Z_P$ 
   $t_7 \leftarrow t_2^2$ 
   $t_5 \leftarrow t_6 - t_7$ 
   $t_3 \leftarrow X_Q + Z_Q$ 
   $t_4 \leftarrow X_Q - Z_Q$ 
   $t_8 \leftarrow t_4 \cdot t_1$ 
   $t_9 \leftarrow t_3 \cdot t_2$ 
   $X_{P+Q} \leftarrow (t_8 + t_9)^2$ 
   $Z_{P+Q} \leftarrow x_{Q-P} \cdot (t_8 - t_9)^2$ 
   $X_{[2]P} \leftarrow t_6 \cdot t_7$ 
   $Z_{[2]P} \leftarrow t_5 \cdot (t_7 + ((A+2)/4) \cdot t_5)$ 
  return ( $X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q}$ )
end function

```

The implementation described here follows [Ber06] and computes x_Q by first using Algorithm 5 to compute (X_Q, Z_Q) and then computing $x_Q = Z_Q^{-1} \cdot X_Q$.

4.2 Implementation of Curve25519

Each Synergistic Processor Unit (SPU) of the Cell Broadband Engine is a specialist processor with a RISC-like SIMD instruction set and a 128-element array of 128-bit vector registers. It has two pipelines (pipeline 0 and pipeline 1); each cycle it can dispatch one instruction per pipeline. Whether or not the SPU really dispatches two instructions in a given cycle is highly dependent on instruction scheduling and alignment. This is subject to the following conditions:

- execution of instructions is purely in-order;
- the two pipelines execute disjoint sets of instructions (i.e. each instruction is either a pipeline-0 or a pipeline-1 instruction);
- the SPU has a fetch queue that can contain at most two instructions;
- instructions are fetched into the fetch queue only if the fetch queue is empty;
- instructions are fetched in pairs, where the first instruction in such a pair is from an even word address, the second from an odd word address;
- the SPU executes two instructions in one cycle only if two instructions are in the fetch queue, the first being a pipeline-0 instruction and the second being a pipeline-1 instruction and all inputs to these instructions being available and not pending due to latencies of previously executed instructions.

Hence, instruction *scheduling* has to ensure that pipeline-0 and pipeline-1 instructions are interleaved and that latencies are hidden; instruction *alignment* has to ensure that pipeline-0 instructions are at even word addresses and pipeline-1 instructions are at odd word addresses.

Most integer-arithmetic instructions (except shift and rotate instructions) are SIMD instructions operating either on 4 32-bit word elements or on 8 16-bit halfword elements or on 16 8-bit byte elements of a 128-bit register. Integer multiplication constitutes an exception to this scheme: The integer multiplication instructions multiply 4 16-bit halfwords in parallel and store the 32-bit results in the 4 word elements of the result register. For a detailed description of the SPU instruction set see [IBM07]; for a list of instruction latencies and associated pipelines see [IBM08, Appendix B]. The most relevant instructions for the implementation of the Curve25519 function are the following:

- a: adds each 32-bit word element of a register *a* to the corresponding word element of a register *b* and stores the results in a register *r*;
- mpy: multiplies the 16 least significant bits of each 32-bit word element of a register *a* with the corresponding 16 bits of each word element of a register *b* and stores the resulting four 32-bit results in the four word elements of a register *r*;
- mpya: multiplies 16-bit halfwords as the mpy instruction but adds the resulting four 32-bit word elements to the corresponding word elements of a register *c* and stores the resulting sum in a register *r*;
- shl: shifts each word element of a register *a* to the left by the number of bits given by the corresponding word element of a register *b* and stores the result in a register *r*;
- rotmi: shifts each word element of a register *a* to the right by the number of bits given in an immediate value and stores the result in a register *r*;
- shufb: each byte of an input register *c* specifies either a byte of one of the two input registers *a* and *b* or one of the constants 0, 0x80 or 0xff. Each byte in the result register *r* is set to the value specified by the corresponding byte in *c*.

As described in Section 4.1 the computation of the Curve25519 function consists of two parts: the Montgomery ladder computing (X_Q, Z_Q) and the inversion of Z_Q . Inversion is implemented as an exponentiation with $p - 2 = 2^{255} - 21$ using the same sequence of 254 squarings and 11 multiplications as [Ber06]. This might not be the most efficient algorithm for inversion, but it is the easiest way to implement an inversion algorithm that takes constant time independent of the input. Completely inlining all multiplications and squarings of the addition chain would result in an excessive increase of the overall code size. Inversion therefore uses calls to a multiplication and a squaring function.

However the first part—the Montgomery ladder—does not use calls to these functions but takes one ladder step as smallest building block and implements the complete Montgomery ladder in one function. This increases data-level parallelism inside

the computation, and allows a much more efficient use of the SIMD instructions, especially in the modular reductions.

The three algorithms that mainly influence performance of the Curve25519 computation are multiplication (including reduction), squaring (also including reduction), and Montgomery ladder steps. These algorithms are implemented in qasm. The description of the algorithms in the following three subsections uses the notion of “register variables”; recall from Section 2.8 that in qasm—unlike C—these are variables that are forced to be kept in registers. The algorithms are all based on a representation of elements of $\mathbb{F}_{2^{255}-19}$ which is chosen to allow efficient implementation with the SIMD instructions of the SPUs.

Specifically, an element a of $\mathbb{F}_{2^{255}-19}$ is represented as a tuple (a_0, \dots, a_{19}) where

$$a = \sum_{i=0}^{19} a_i 2^{\lceil 12.75i \rceil}. \quad (4.1)$$

A coefficient a_i is called *reduced* if $a_i \in [0, 2^{13} - 1]$. Analogously the representation of an element $a \in \mathbb{F}_{2^{255}-19}$ is called reduced if all its coefficients a_0, \dots, a_{19} are reduced.

4.2.1 Multiplication and squaring

The SPUs of the Cell Broadband Engine can multiply integers of size at most 16 bits, yielding 32-bit results. One instruction performs 4 such multiplications in parallel. In order to achieve high performance of finite-field arithmetic it is crucial to properly arrange the values a_0, \dots, a_{19} in registers and to adapt algorithms for field arithmetic to make use of this SIMD capability.

The inputs of field multiplication are two finite-field elements (a_0, \dots, a_{19}) and (b_0, \dots, b_{19}) in reduced representation. This input is arranged in 10 register variables a03, a47, a811, a1215, a1619, b03, b47, b811, b1215 and b1619 as follows: Register variable a03 contains in its word elements the coefficients a_0, a_1, a_2, a_3 , register variable a47 contains in its word elements the coefficients a_4, a_5, a_6, a_7 , and so on.

The idea of multiplication is to compute coefficients r_0, \dots, r_{38} of $r = ab$ where:

$$\begin{aligned} r_0 &= a_0 b_0 \\ r_1 &= a_1 b_0 + a_0 b_1 \\ r_2 &= a_2 b_0 + a_1 b_1 + a_0 b_2 \\ r_3 &= a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 \\ r_4 &= a_4 b_0 + 2a_3 b_1 + 2a_2 b_2 + 2a_1 b_3 + a_0 b_4 \\ r_5 &= a_5 b_0 + a_4 b_1 + 2a_3 b_2 + 2a_2 b_3 + a_1 b_4 + a_0 b_5 \\ r_6 &= a_6 b_0 + a_5 b_1 + a_4 b_2 + 2a_3 b_3 + a_2 b_4 + a_1 b_5 + a_0 b_6 \\ r_7 &= a_7 b_0 + a_6 b_1 + a_5 b_2 + a_4 b_3 + a_3 b_4 + a_2 b_5 + a_1 b_6 + a_0 b_7 \\ r_8 &= a_8 b_0 + 2a_7 b_1 + 2a_6 b_2 + 2a_5 b_3 + a_4 b_4 + 2a_3 b_5 + 2a_2 b_6 + 2a_1 b_7 + a_0 b_8 \\ &\vdots \end{aligned}$$

This computation requires 400 multiplications and 361 additions. At best 4 of these multiplications can be done in parallel when using the SIMD instructions, and adding the result of a multiplication is at best for free using the `mpya` instruction, so it takes at least 100 instructions to compute the coefficients r_0, \dots, r_{38} . Furthermore some intermediate products have to be multiplied by 2, an effect resulting from the non-integer radix 12.75 used for the representation of finite-field elements. As both inputs to multiplication have reduced coefficients, all result coefficients r_i fit into 32-bit word elements.

The coefficients r_0, \dots, r_{38} can be computed using 145 pipeline-0 instructions (arithmetic instructions) as described in the following three paragraphs. This computation requires some rearrangement of coefficients in registers using the `shufb` instruction but with careful instruction scheduling and alignment these pipeline-1 instructions do not increase the number of cycles needed for multiplication. From the description of the arithmetic instructions it should be clear which rearrangement of inputs is necessary.

First use 15 `shl` instructions to obtain register variables

`b03s1` containing $b_0, b_1, b_2, 2b_3$,
`b03s2` containing $b_0, b_1, 2b_2, 2b_3$,
`b03s3` containing $b_0, 2b_1, 2b_2, 2b_3$,
`b47s1` containing $b_4, b_5, b_6, 2b_7$ and so on.

Now proceed producing intermediate result variables

`r03` containing $a_0b_0, a_0b_1, a_0b_2, a_0b_3$ (one `mpy` instruction),
`r14` containing $a_1b_0, a_1b_1, a_1b_2, 2a_1b_3$ (one `mpy` instruction),
`r25` containing $a_2b_0, a_2b_1, 2a_2b_2, 2a_2b_3$ (one `mpy` instruction),
`r36` containing $a_3b_0, 2a_3b_1, 2a_3b_2, 2a_3b_3$ (one `mpy` instruction),
`r47` containing $a_4b_0 + a_0b_4, a_4b_1 + a_0b_5, a_4b_2 + a_0b_6, a_4b_3 + a_0b_7$ (one `mpy` and one `mpya` instruction),
`r58` containing $a_5b_0 + a_1b_4, a_5b_1 + a_1b_5, a_5b_2 + a_1b_6, 2a_5b_3 + 2a_1b_7$ (one `mpy` and one `mpya` instruction) and so on. In total these computations need 36 `mpy` and 64 `mpya` instructions.

As a final step these intermediate results have to be joined to produce the coefficients r_0, \dots, r_{38} in the register variables `r03, r47, ... r3639`. This can be done using 30 additions when first combining intermediate results using the `shufb` instruction. For example, join in one register variable the highest word of `r14` and the three lowest words of `r58` before adding this register variable to `r47`.

The basic idea for squaring is the same as for multiplication, but squaring can exploit the fact that some intermediate results are equal. For a squaring of a value a given in reduced representation (a_0, \dots, a_{19}) , formulas for the result coefficients r_0, \dots, r_{38} are the following:

$$\begin{aligned}
r_0 &= a_0 a_0 \\
r_1 &= 2a_1 a_0 \\
r_2 &= 2a_2 a_0 + a_1 a_1 \\
r_3 &= 2a_3 a_0 + 2a_2 a_1 \\
r_4 &= 2a_4 a_0 + 4a_3 a_1 + 2a_2 a_2 \\
r_5 &= 2a_5 a_0 + 2a_4 a_1 + 4a_3 a_2 \\
r_6 &= 2a_6 a_0 + 2a_5 a_1 + 2a_4 a_2 + 2a_3 a_3 \\
r_7 &= 2a_7 a_0 + 2a_6 a_1 + 2a_5 a_2 + 2a_4 a_3 \\
r_8 &= 2a_8 a_0 + 4a_7 a_1 + 4a_6 a_2 + 4a_5 a_3 + a_4 a_4 \\
&\vdots
\end{aligned}$$

The main part of the computation requires only 60 multiplications (24 mpya and 36 mpy instructions). However, some partial results have to be multiplied by 4; this requires more preprocessing of the inputs than for multiplication, in total 35 instead of 15 shl instructions before entering the main block of multiplications. Squaring is therefore only 20 cycles faster than multiplication.

During both multiplication and squaring, almost all latencies are hidden by interleaving independent instructions.

4.2.2 Reduction

The task of the reduction step is to compute from the coefficients r_0, \dots, r_{38} a reduced representation (r_0, \dots, r_{19}) . Implementing this computation efficiently is challenging because in a typical reduction chain every instruction is dependent on the result of the preceding instruction. This makes it very hard to vectorize operations in SIMD instructions and to hide latencies.

The basic idea of reduction is the following: First reduce the coefficients r_{20} to r_{38} (producing a coefficient r_{39}), then add $19r_{20}$ to r_0 , $19r_{21}$ to r_1 and so on until adding $19r_{39}$ to r_{19} and then reduce the coefficients r_0 to r_{19} .

Multiplications by 19 result from the fact that the coefficient a_{20} corresponds to $2^{255}a_{20}$ (see equation (4.1)). By the definition of the finite field $\mathbb{F}_{2^{255}-19}$, $2^{255}a_{20}$ is the same as $19a_{20}$. Equivalent statements hold for the coefficients a_{21}, \dots, a_{39} .

The most speed-critical parts of this reduction are the two carry chains from r_{20} to r_{39} and from r_0 to r_{19} . Latencies in these chains can be hidden if each chain is broken into four parallel carry chains; Algorithm 7 describes this structure of the modular reduction algorithm. Each of the carry operations in Algorithm 7 can be done using one shufb, one rotmi and one a instruction. Furthermore each of the two carry chains requires 8 masking instructions (bitwise and).

In total, a call to the multiplication function (including reduction) takes 444 cycles, and a call to the squaring function takes 424 cycles. This includes 144 cycles for multiplication (124 cycles for squaring), 244 cycles for reduction and some more cy-

Algorithm 7 Structure of the modular reduction in $\mathbb{F}_{2^{255}-19}$

Carry from r_{20} to r_{21} , from r_{24} to r_{25} , from r_{28} to r_{29} and from r_{32} to r_{33}
 Carry from r_{21} to r_{22} , from r_{25} to r_{26} , from r_{29} to r_{30} and from r_{33} to r_{34}
 Carry from r_{22} to r_{23} , from r_{26} to r_{27} , from r_{30} to r_{31} and from r_{34} to r_{35}
 Carry from r_{23} to r_{24} , from r_{27} to r_{28} , from r_{31} to r_{32} and from r_{35} to r_{36}

Carry from r_{24} to r_{25} , from r_{28} to r_{29} , from r_{32} to r_{33} and from r_{36} to r_{37}
 Carry from r_{25} to r_{26} , from r_{29} to r_{30} , from r_{33} to r_{34} and from r_{37} to r_{38}
 Carry from r_{26} to r_{27} , from r_{30} to r_{31} , from r_{34} to r_{35} and from r_{38} to r_{39}
 Carry from r_{27} to r_{28} , from r_{31} to r_{32} and from r_{35} to r_{36}

Add $19r_{20}$ to r_0 , add $19r_{21}$ to r_1 , add $19r_{22}$ to r_2 and add $19r_{23}$ to r_3
 Add $19r_{24}$ to r_4 , add $19r_{25}$ to r_5 , add $19r_{26}$ to r_6 and add $19r_{27}$ to r_7
 Add $19r_{28}$ to r_8 , add $19r_{29}$ to r_9 , add $19r_{30}$ to r_{10} and add $19r_{31}$ to r_{11}
 Add $19r_{32}$ to r_{12} , add $19r_{33}$ to r_{13} , add $19r_{34}$ to r_{14} and add $19r_{35}$ to r_{15}
 Add $19r_{36}$ to r_{16} , add $19r_{37}$ to r_{17} , add $19r_{38}$ to r_{18} and add $19r_{39}$ to r_{19}

Carry from r_{16} to r_{17} , from r_{17} to r_{18} , from r_{18} to r_{19} and from r_{19} to r_{20}
 Add $19r_{20}$ to r_0

Carry from r_0 to r_1 , from r_4 to r_5 , from r_8 to r_9 and from r_{12} to r_{13}
 Carry from r_1 to r_2 , from r_5 to r_6 , from r_9 to r_{10} and from r_{13} to r_{14}
 Carry from r_2 to r_3 , from r_6 to r_7 , from r_{10} to r_{11} and from r_{14} to r_{15}
 Carry from r_3 to r_4 , from r_7 to r_8 , from r_{11} to r_{12} and from r_{15} to r_{16}

Carry from r_4 to r_5 , from r_8 to r_9 , from r_{12} to r_{13} and from r_{16} to r_{17}
 Carry from r_5 to r_6 , from r_9 to r_{10} , from r_{13} to r_{14} and from r_{17} to r_{18}
 Carry from r_6 to r_7 , from r_{10} to r_{11} , from r_{14} to r_{15} and from r_{18} to r_{19}
 Carry from r_7 to r_8 , from r_{11} to r_{12} and from r_{15} to r_{16}

cles to load input and store output. Furthermore the cost of a function call is included in these numbers.

4.2.3 Montgomery ladder step

The implementation of a Montgomery ladder step optimizes a fixed sequence of arithmetic instructions in $\mathbb{F}_{2^{255}-19}$ instead of single instructions. This makes it much easier to make efficient use of the SIMD instruction set.

The idea is to arrange the operations in $\mathbb{F}_{2^{255}-19}$ into blocks of 4 equal or similar instructions, “similar” meaning that multiplications and squarings can be grouped together and additions and subtractions can be grouped together as well. Then these operations can be carried out using the 4-way parallel SIMD instructions in the obvious way; for example for 4 multiplications $r = a \cdot b$, $s = c \cdot d$, $t = e \cdot f$ and $u = g \cdot h$ first produce register variables `aceg0` containing in its word elements a_0, c_0, e_0, g_0

and `bdgh0` containing b_0, d_0, e_0, g_0 and so on. Then the first coefficient of r, s, t and u can be computed by applying the `mpy` instruction on `aceg0` and `bdfh0`. All other result coefficients of r, s, t and u can be computed in a similar way using `mpy` and `mpya` instructions.

For this way of using the SIMD capabilities of CPUs [GGP09] introduced the term “digitslicing”. This approach not only makes multiplication slightly faster (420 arithmetic instructions instead of 576 for 4 multiplications), it also makes it possible to vectorize modular reduction: The reduction algorithm described above can now be applied to 4 results in parallel, reducing the cost of a reduction by a factor of 4.

Algorithm 8 describes how a Montgomery ladder step is divided into blocks of 4 similar operations. In this algorithm the computation of Z_{p+Q} in the last step requires one multiplication and reduction which is carried out as described in the previous section. The computation of a ladder step again requires rearrangement of data in registers using the `shufb` instruction, but these pipeline-1 instructions can almost entirely be hidden by interleaving them with arithmetic pipeline-0 instructions.

One remark regarding subtractions occurring in this computation: As reduction expects all coefficients to be larger than zero, subtraction cannot simply compute the difference of each coefficient. To subtract b from a one must instead first add $2p$ to a and then subtract b . For blocks containing additions and subtractions in Algorithm 8 the additions are computed together with additions of $2p$ the subtractions are performed in a separate step.

To minimize function-call overhead, also loop control for the Montgomery ladder is implemented in `qasm` yielding one function for the whole Montgomery ladder. One call to this function involving 255 Montgomery ladder steps takes 582720 cycles.

Algorithm 8 Structure of a Montgomery ladder step (see Algorithm 6) optimized for 4-way parallel computation

function LADDERSTEP($x_{Q-P}, X_P, Z_P, X_Q, Z_Q$)

$$t_1 \leftarrow X_P + Z_P$$

$$t_2 \leftarrow X_P - Z_P$$

$$t_3 \leftarrow X_Q + Z_Q$$

$$t_4 \leftarrow X_Q - Z_Q$$

Reduce t_1, t_2, t_3, t_4

$$t_6 \leftarrow t_1 \cdot t_1$$

$$t_7 \leftarrow t_2 \cdot t_2$$

$$t_8 \leftarrow t_4 \cdot t_1$$

$$t_9 \leftarrow t_3 \cdot t_2$$

Reduce t_6, t_7, t_8, t_9

$$t_{10} \leftarrow ((A+2)/4) \cdot t_6$$

$$t_{11} \leftarrow ((A+2)/4 - 1) \cdot t_7$$

$$t_5 \leftarrow t_6 - t_7$$

$$t_4 \leftarrow t_{10} - t_{11}$$

$$t_1 \leftarrow t_8 - t_9$$

$$t_0 \leftarrow t_8 + t_9$$

Reduce t_5, t_4, t_1, t_0

$$Z_{[2]P} \leftarrow t_5 \cdot t_4$$

$$X_{P+Q} \leftarrow t_0^2$$

$$X_{[2]P} \leftarrow t_6 \cdot t_7$$

$$t_2 \leftarrow t_1 \cdot t_1$$

Reduce $Z_{[2]P}, X_{P+Q}, X_{[2]P}, t_2$

$$Z_{P+Q} \leftarrow x_{Q-P} \cdot t_2$$

Reduce Z_{P+Q}

return ($X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q}$)

end function

4.3 Performance results and comparison

The implementation described in this chapter is the first—and to date the only—high-speed implementation of ECC for the Cell Broadband Engine. There exists some literature describing related work but no high-speed implementation of ECC at a similar security level which would allow a direct comparison of the achieved speeds.

Bos in [Bos10] describes an implementation of parallel finite-field multiplication including multiplication in the field $\mathbb{F}_{2^{255}-19}$. For 8-way parallel digit-sliced multiplication he achieves 180 cycles per multiplication, similar to the multiplication speed achieved in the 4-way parallel multiplication inside a Montgomery ladder step described in Subsection 4.2.3.

A cluster of PlayStations has been used in [BKM09] to solve an ECDLP on a curve over a 112-bit prime field. A faster implementation to solve the same ECDLP targeting the same architecture is described in [BLS10]. These implementations cannot be compared to the implementation described in this chapter; not only are the field sizes smaller and the target function quite different, the cryptanalytical algorithms have a much higher level of parallelism and make it much easier to exploit the SIMD instruction set of the SPUs. However, the implementation presented in [BLS10] uses several ideas of the implementation described here.

Previous implementations of public-key cryptography for the Cell Broadband Engine include an implementation of RSA by Scott and Costigan [CS07]. The paper reports benchmarks for RSA with different key lengths; RSA signature generation with a 2048 bit key is reported to take 0.015636s corresponding to 50,035,200 cycles on one SPU. Note that although 2048-bit RSA has a significantly lower security level than the Curve25519 function, this is more than 7 times slower than the Curve25519 implementation described in this chapter.

The benchmark results of the Curve25519 implementation are obtained using the SUPERCOP benchmarking framework [BLa]. The software passes all tests of this toolkit showing compatibility to other Curve25519 implementations, in particular the reference implementation included in the toolkit. The computers used for benchmarking are described in Table 4.1.

For scalar-multiplication software, the SUPERCOP benchmarking suite measures two different cycle counts: The `crypto_scalarmult` benchmark measures cycles for a scalar multiplication of an arbitrary point; the `crypto_scalarmult_base` benchmark measures cycles needed for a scalar multiplication of a fixed base point. The software presented in this chapter does not contain a dedicated function for multiplication of a fixed base point; faster implementations using precomputed multiples of the base point would be useful in applications that frequently call the function `crypto_scalarmult_base`. Two further benchmarks measure the Curve25519 software in the context of Diffie-Hellman key exchange: The `crypto_dh_keypair` benchmark measures the number of cycles to generate a key pair consisting of a secret and a public key. In addition to the scalar multiplication in the Curve25519 function this requires generating a random scalar. The `crypto_dh` benchmark measures cycles to compute a joint key, given a secret and a public key. The results of all benchmarks are given in Table 4.2.

Table 4.1: Machines used for benchmarking the Curve25519 implementation for Cell processors

hex01		
Affiliation	RWTH Aachen University	
CPU	Cell Broadband Engine rev 5.1, 3200 MHz	
Compilers	gcc spu-gcc	gcc (GCC) 4.1.2 20070925 (Red Hat 4.1.2-27) spu-gcc (GCC) 4.1.1
s1c1b2		
Affiliation	Barcelona Supercomputing Center	
CPU	PowerXCell 8i (Cell Broadband Engine rev. 48), 3200 MHz	
Compilers	gcc gcc34 gcc43 spu-gcc	gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-44) gcc34 (GCC) 3.4.6 20060404 (Red Hat 3.4.6-4) gcc43 (GCC) 4.3.2 20081007 (Red Hat 4.3.2-7) spu-gcc (GCC) 4.1.1
stan		
Affiliation	Peter Schwabe	
CPU	Cell Broadband Engine (rev 5.1), 3192 MHz	
Compilers	gcc spu-gcc	gcc (GCC) 4.4.4 20100630 (Red Hat 4.4.4-10) spu-gcc (GCC) 4.1.1

Table 4.2: Cycle counts of the Curve25519 software on different machines

SUPERCOP benchmark	hex01	node001	stan
crypto_scalarmult	697440 ^a	696240 ^c	695640 ^d
crypto_scalarmult_base	697440 ^a	696240 ^c	695720 ^d
crypto_dh_keypair	720240 ^b	716280 ^a	720880 ^e
crypto_dh	696960 ^b	696240 ^a	696760 ^e

^a Compiled with spu-gcc -funroll-loops -mstdmain -march=cell -O3 -fomit-frame-pointer -Drandom=rand -Dsrandom=srand

^b Compiled with spu-gcc -mstdmain -march=cell -O3 -fomit-frame-pointer -Drandom=rand -Dsrandom=srand

^c Compiled with spu-gcc -funroll-loops -mstdmain -march=celledp -O2 -fomit-frame-pointer -Drandom=rand -Dsrandom=srand

^d Compiled with spu-gcc -funroll-loops -mstdmain -march=cell -O -fomit-frame-pointer -Drandom=rand -Dsrandom=srand

^e Compiled with spu-gcc -mstdmain -march=cell -Os -fomit-frame-pointer -Drandom=rand -Dsrandom=srand

5

Pairing computation on AMD64 processors

Cryptographic pairings are a building block used for various cryptographic protocols, for example short-signature schemes [BLS01, BLS04], identity-based encryption schemes [BF01, BF03], searchable cryptography [BCOP04], and non-interactive zero-knowledge proofs [GS08]. Cryptographic pairings in practice are based on arithmetic on elliptic curves. Making pairings secure and efficiently computable requires very special elliptic curves, so called pairing-friendly curves.

This chapter describes an implementation of a cryptographic pairing on a Barreto-Naehrig curve for AMD64 processors. The implementation is more than two times faster than all previously published implementations. It is furthermore the first implementation that exploits the special structure of the underlying field for faster arithmetic in software. However, a followup implementation by Beuchat et al. achieved even higher speeds for a cryptographic pairing over a Barreto-Naehrig curve at a similar security level in [BGM⁺10].

Section 5.1 gives a short general background on cryptographic pairings. Section 5.2 gives mathematical and algorithmic details of the optimal ate pairing over Barreto-Naehrig curves used for the implementation. The description of implementation details is split into three sections: Section 5.3 describes high-level optimizations and algorithms used for pairing computation, Section 5.4 explains the representation of elements of the underlying finite field and Section 5.5 details how field-arithmetic operations are implemented on assembly level. Section 5.6 gives performance results and a comparison with previous implementations. Furthermore it analyzes why the followup implementation in [BGM⁺10] achieved even higher speeds.

This chapter is based on joint work with Naehrig and Niederhagen published in [NNS10]. Aside from some minor changes to notation and phrasing the main differences between the content of this chapter and [NNS10] are the following:

- [NNS10] contains an error in the choice of the elliptic curve. This error is corrected here and in online versions (since version 2010-07-14) of the paper.
- Several aspects of the implementation improved since [NNS10] was published, these improvements are described here, the benchmarks in Section 5.6 are updated accordingly;
- this chapter has a different introduction, the comparison to previous work is contained in Section 5.6;
- the general background on cryptographic pairings in Section 5.1 is not contained in [NNS10], it is using material from the introduction of [NNS10] and of [KZS⁺09].
- Section 5.6 gives a detailed comparison with the implementation presented in [BGM⁺10], this comparison is not contained in [NNS10].

Parts of earlier versions of the software presented here were also used in [NBS08] and [KZS⁺09].

5.1 Background on cryptographic pairings

This section only gives a short introduction to cryptographic pairings. For a much more comprehensive introduction see for example [BSS05, chapter IX] or [Nae09, Chapter 1].

For three groups G_1 , G_2 (written additively) and G_3 (written multiplicatively) of prime order r a cryptographic pairing is a map $e : G_1 \times G_2 \rightarrow G_3$ with the following properties:

- bilinearity:

$$e(mP, Q) = e(P, mQ) = e(P, Q)^m \text{ for } m \in \mathbb{Z};$$

- non-degeneracy:

there exist $P \in G_1$ and $Q \in G_2$ such that $e(P, Q) \neq 1$;

- computability:

there exists an efficient algorithm to compute $e(P, Q)$ given P and Q .

In practice, cryptographic pairings are usually constructed as follows: Let E be an elliptic curve defined over a finite field \mathbb{F}_q . Let r be a prime dividing the group order $\#E(\mathbb{F}_q) = n$ and let k be the smallest integer such that $r \mid q^k - 1$. The integer k is called the *embedding degree of E with respect to r* .

Let $P_0 \in E(\mathbb{F}_q)$ and $Q_0 \in E(\mathbb{F}_{q^k})$ be points of order r such that $Q_0 \notin \langle P_0 \rangle$. Let $\mathcal{O} \in E(\mathbb{F}_q)$ denote the neutral element of the group operation on E . Define $G_1 = \langle P_0 \rangle$ and $G_2 = \langle Q_0 \rangle$. Let $G_3 = \mu_r$ be the group of r -th roots of unity in $\mathbb{F}_{q^k}^*$.

For $i \in \mathbb{Z}$ and $P \in E$ a Miller function [Mil86a, Mil04] is an element $f_{i,P}$ of the function field of E such that the divisor of $f_{i,P}$ is

$$\operatorname{div}(f_{i,P}) = i(P) - ([i]P) - (i-1)\mathcal{O},$$

where $[m]$ denotes the multiplication-by- m map on E . The *reduced Tate pairing* τ is defined in terms of Miller functions as

$$\tau : G_1 \times G_2 \rightarrow \mu_r; \quad (P, Q) \mapsto f_{r,P}(Q)^{(q^k-1)/r}.$$

Several variants of the Tate pairing have been proposed such as the eta, ate and twisted ate pairings [BGOS07, HSV06], the R-ate [LLP09] and optimal ate pairings [Ver10] (see also [Bar10]), often increasing computational efficiency over that of their predecessors. Some of these variants, including the optimal ate pairings, require switching the arguments.

Pairing-based protocols involve elliptic-curve groups as well as subgroups of the multiplicative group of a finite field. All pairing-based protocols require that the discrete-logarithm problem (DLP) is hard in all groups. The hardness of the DLP and efficiency of arithmetic in $\mathbb{F}_{q^k}^*$ highly depend on the embedding degree k and consequently several methods have been proposed to construct cryptographically secure *pairing-friendly curves* with predefined embedding degree [FST10]. At the 128-bit security level, the best choice for a pairing-friendly curve is a Barreto-Naehrig curve (BN curve) [BN06] over a prime field \mathbb{F}_p of size roughly 256 bits with embedding degree $k = 12$; for details on Barreto-Naehrig curves also see [Nae09, Chapter 2]. According to [BBB⁺07] such a curve achieves 128 bits of security, according to [ECR09] only 124 bits.

5.2 An optimal ate pairing on Barreto-Naehrig curves

For a BN curve, the most efficient pairings are the R-ate pairing [LLP09] and optimal ate pairings [Ver10]. This section provides basic background and notation for BN curves, and describes the algorithm for the optimal ate pairing that is used in the implementation.

Let u be an integer such that

$$\begin{aligned} p = p(u) &= 36u^4 + 36u^3 + 24u^2 + 6u + 1, \text{ and} \\ n = n(u) &= 36u^4 + 36u^3 + 18u^2 + 6u + 1, \end{aligned}$$

are both prime. An elliptic curve E over \mathbb{F}_p with $|E(\mathbb{F}_p)| = n$ is called a Barreto-Naehrig curve (BN curve). According to [Nae09, Theorem 2.2] such a curve exists, has the form $E : y^2 = x^3 + b$, and embedding degree $k = 12$ with respect to n .

Let ϕ_p be the p -power Frobenius endomorphism on E and $E[n]$ the n -torsion subgroup of E . Define

$$\begin{aligned} G_1 &= E[n] \cap \ker(\phi_p - [1]) = E(\mathbb{F}_p), \\ G_2 &= E[n] \cap \ker(\phi_p - [p]) \subseteq E(\mathbb{F}_{p^{12}})[n], \text{ and} \\ G_3 &= \mu_n, \end{aligned}$$

where $\mu_n \subset \mathbb{F}_{p^{12}}^*$ is the group of n -th roots of unity.

An optimal ate pairing on E is given in [Ver10] as

$$a_{\text{opt}} : G_2 \times G_1 \rightarrow G_3, (Q, P) \mapsto (f_{6u+2,Q}(P) \cdot g_{6u+2,Q}(P))^{(p^{12}-1)/n},$$

where $g_{6u+2,Q}(P) = l_{Q_3, -Q_2}(P) \cdot l_{-Q_2+Q_3, Q_1}(P) \cdot l_{Q_1-Q_2+Q_3, [6u+2]Q}(P)$ with $Q_1 = \phi_p(Q)$, $Q_2 = \phi_p^2(Q)$, and $Q_3 = \phi_p^3(Q)$. The value $l_{R,S}(P) \in \mathbb{F}_{p^{12}}$ is the function of the line through the points R and S on the curve, evaluated at P .

There is no need to compute Q_3 . Instead, $g_{6u+2,Q}(P)$ can be replaced by

$$h_{6u+2,Q}(P) = l_{[6u+2]Q, Q_1}(P) \cdot l_{[6u+2]Q+Q_1, -Q_2}(P).$$

The reason for this is that for BN curves $Q_1 - Q_2 + Q_3 + [6u+2]Q = \mathcal{O}$, which can be easily derived from [Nae09, Lemma 2.17]. By writing down the divisors of the functions $g_{6u+2,Q}$ and $h_{6u+2,Q}$, it can be seen that they differ only by vertical line functions. When evaluated at P , such line functions produce values in proper subfields of $\mathbb{F}_{p^{12}}$ that are mapped to 1 when raised to the power of $(p^{12}-1)/n$.

Algorithm 9 Optimal ate pairing on BN curves for $u > 0$

Input: $P \in G_1, Q \in G_2, m_{\text{opt}} = 6u + 2 = (1, m_{s-1}, \dots, m_0)_{\text{NAF}}, m_i \in \{-1, 0, 1\}$.

Output: $a_{\text{opt}}(Q, P)$.

```

1:  $R \leftarrow Q, f \leftarrow 1$ 
2: for  $i \leftarrow s - 1$  downto 0 do
3:    $f \leftarrow f^2 \cdot l_{R,R}(P), R \leftarrow [2]R$ 
4:   if  $(m_i = -1)$  then
5:      $f \leftarrow f \cdot l_{R,-Q}(P), R \leftarrow R - Q$ 
6:   else if  $(m_i = 1)$  then
7:      $f \leftarrow f \cdot l_{R,Q}(P), R \leftarrow R + Q$ 
8:   end if
9: end for
10:  $Q_1 = \phi_p(Q), Q_2 = \phi_p^2(Q)$ 
11:  $f \leftarrow f \cdot l_{R, Q_1}(P), R \leftarrow R + Q_1$ 
12:  $f \leftarrow f \cdot l_{R, -Q_2}(P), R \leftarrow R - Q_2$ 
13:  $f \leftarrow f^{p^6-1}$ 
14:  $f \leftarrow f^{p^2+1}$ 
15:  $f \leftarrow f^{(p^4-p^2+1)/n}$ 
16: return  $f$ 

```

Algorithm 9 shows how $a_{\text{opt}}(Q, P)$ can be computed in the case $u > 0$ and when m_{opt} is given in non-adjacent form (NAF). Lines 2 to 9 are called the Miller loop; it contains doubling steps in Line 3 and addition steps in Lines 5 and 7. The result of the Miller loop is multiplied by $h_{6u+2, Q}(P)$ through two additions steps in Lines 10 to 12. Lines 13 to 15 together carry out the final exponentiation to the power $(p^{12} - 1)/n$, where Lines 13 and 14 comprise the so-called easy part of the exponentiation. This easy part can be done by one application of the p^6 -power Frobenius automorphism on $\mathbb{F}_{p^{12}}$, one application of the p^2 -Frobenius automorphism, a single inversion and two multiplications in $\mathbb{F}_{p^{12}}$. Line 15 represents the hard part of the final exponentiation.

As usual [BN06, HSV06, DSD07], points in G_2 are represented by points on the sextic twist $E' : y^2 = x^3 + b/\xi$ defined over \mathbb{F}_{p^2} using the twist isomorphism $\psi : E' \rightarrow E$, $(x', y') \mapsto (\omega^2 x', \omega^3 y')$. The element $\xi \in \mathbb{F}_{p^2}$ (neither a cube nor a square in \mathbb{F}_{p^2}) is chosen such that the twist has the right order, i.e. it holds $n \mid \#E'(\mathbb{F}_{p^2})$. The field $\mathbb{F}_{p^{12}}$ is generated over \mathbb{F}_{p^2} by ω via the irreducible polynomial $X^6 - \xi$, i.e. $\omega^6 = \xi$.

The map ψ induces a group isomorphism between $G'_2 = E'(\mathbb{F}_{p^2})[n]$ and G_2 . So, all points $R \in G_2$ should be seen as being represented by a corresponding point $R' \in G'_2$, i.e. $R = \psi(R')$. All curve arithmetic is done on the twist and intermediate results can be kept in their representation on E' . This means that all curve arithmetic requires \mathbb{F}_{p^2} -arithmetic only. Arithmetic in $\mathbb{F}_{p^{12}}$ is also based on arithmetic in \mathbb{F}_{p^2} . Overall, there are no \mathbb{F}_p computations in the optimal-ate pairing algorithm other than those involved in \mathbb{F}_{p^2} computations. Thus an improvement of \mathbb{F}_{p^2} -arithmetic—even without improving \mathbb{F}_p -arithmetic—leads to an improvement of all parts of the computation.

5.3 High-level techniques

This section describes the high-level structure of the implementation, which uses state-of-the-art optimization techniques from the literature. The focus is on the construction of the higher-degree field extensions such as \mathbb{F}_{p^6} and $\mathbb{F}_{p^{12}}$, the Miller loop to compute $f_{6u+2, Q}(P)$, the function value $h_{6u+2, Q}(P)$ and the structure of the final exponentiation.

The construction of field extensions and the efficiency of the optimal ate pairing depend on the chosen curve parameters. Since the representation of elements in \mathbb{F}_p described in Section 5.4 needs the parameter u to be a third power $u = v^3$, the choice of a curve is strongly restricted. Field multiplications and squarings in $\mathbb{F}_{p^{12}}$ are very expensive, so another important condition on u is that the NAF representation of $6u + 2$ should have as few non-zero entries as possible, to save as many addition steps during the Miller loop as possible.

The specific choice here is $u = v^3$ with $v = 1868033$. This value provides a low NAF weight for $6u + 2$ together with a good addition-subtraction chain for u -th powering for use in the final exponentiation.

5.3.1 Field extensions

The above choice for u implies $p \equiv 3 \pmod{4}$ which means that the field extension \mathbb{F}_{p^2} can be constructed as $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, where $i^2 = -1$. The value ξ to construct the twist and higher-degree extensions is $\xi = i + 3$.

On top of the quadratic extension the field $\mathbb{F}_{p^{12}}$ is built as a tower, first $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}(\tau)$ with $\tau^3 = \xi$ and then $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}(\omega)$ with $\omega^2 = \tau$. This is the same construction as in [DSD07] and the implementation of the extension-field arithmetic follows [DSD07].

5.3.2 Miller loop

The value $6u + 2$ determines the number of doubling and addition steps in the Miller loop of the optimal ate pairing. The number of doubling steps is 65. The NAF weight of $6u + 2$ is 19, so there are 18 addition steps. Throughout the pairing computation points in G_2 are represented as elements of G'_2 in Jacobian coordinates. In particular, for the doubling and addition steps, the implementation uses the formulas given by Arène et al. in [ALNR09]. The points in G_1 , at which line functions are evaluated, are kept in affine coordinates.

The multiplication of the intermediate variable f with the line-function value in the Miller loop is done via a special multiplication function exploiting the fact that line-function values are sparse elements of $\mathbb{F}_{p^{12}}$, where only half of the coefficients over \mathbb{F}_{p^2} are different from zero.

After the Miller loop, the points Q_1 and Q_2 are computed by applying the p -power and the p^2 -power Frobenius endomorphisms. Two final addition steps, with Q_1 and $-Q_2$, respectively, multiply the result of the Miller loop by the function value $h_{6u+2,Q}(P)$.

5.3.3 Final exponentiation

The final exponentiation is done as indicated in Lines 13 to 15 of Algorithm 9. It is divided into the easy part (Lines 13, 14) and the hard part (Line 15). The easy part has low computational costs compared to the hard part: Raising the element to the power $p^6 - 1$ is simply a conjugation in the extension $\mathbb{F}_{p^{12}}/\mathbb{F}_{p^6}$ and a single division in $\mathbb{F}_{p^{12}}$. The exponentiation by $p^2 + 1$ is done by applying the p^2 -power Frobenius automorphism and one multiplication.

Note that after the easy part of the final exponentiation, the resulting element in $\mathbb{F}_{p^{12}}$ lies in the cyclotomic subgroup of $\mathbb{F}_{p^{12}}^*$, i.e. the subgroup of order $\Phi_{12}(p) = p^4 - p^2 + 1$, where Φ_{12} is the 12-th cyclotomic polynomial. Granger and Scott [GS10] showed how to exploit this fact to obtain very efficient squaring formulas for such elements; these formulas are used during the hard part of the final exponentiation. Furthermore, the inverse of an element in the cyclotomic subgroup can be computed by a conjugation over \mathbb{F}_{p^6} , which means that inversions are essentially for free.

The hard part of the final exponentiation uses the method proposed by Scott et al. in [SBC⁺09]. The main advantage here is that the exponentiation is essentially

split into three exponentiations by the exponent $u = v^3$, each implemented as three exponentiations by v . The latter can be done with 21 special squarings and 4 multiplications in $\mathbb{F}_{p^{12}}$ using the addition-subtraction chain described in Algorithm 10. In the end, the final result is obtained by applying the Frobenius automorphism and by using the polynomial representation in u of the fixed exponent $(p^4 - p^2 + 1)/n$.

Algorithm 10 Exponentiation by $v = 1868033$

Input: $a \in \mathbb{F}_{p^{12}}$.

Output: a^v .

1: $t_0 \leftarrow a^8$
 2: $t_1 \leftarrow t_0^8$
 3: $t_2 \leftarrow t_0^{-1}$
 4: $t_2 \leftarrow t_2 \cdot a$
 5: $t_2 \leftarrow t_2 \cdot t_1$
 6: $t_2 \leftarrow t_2^{128}$
 7: $t_2 \leftarrow t_2 \cdot a$
 8: $t_2 \leftarrow t_2^{256}$
 9: $t_2 \leftarrow t_2 \cdot a$
 10: **return** t_2

5.4 Mid-level techniques: arithmetic in \mathbb{F}_{p^2} and \mathbb{F}_p

This section explains the new approach for representing integers modulo p where p is given by the BN polynomial $36u^4 + 36u^3 + 24u^2 + 6u + 1$. An integer mod p is split into 12 coefficients each of which will be stored in a double-precision floating-point variable in the software implementation. This representation is similar to and inspired by the representation used in Bernstein's implementation of Curve25519 [Ber06],

5.4.1 Representing base field elements

Elements in the base field \mathbb{F}_p are integers modulo the prime $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ for some $u \in \mathbb{Z}$. The technique in this section does not depend on the specific fixed value for u used in the implementation. The only requirement is that there exists an integer $v \in \mathbb{Z}$ with $u = v^3$. Furthermore, let $\delta = \sqrt[6]{6}$, so that $(\delta vx)^3 = \sqrt{6}ux^3$.

Integers are represented by polynomials in the ring

$$R = \mathbb{Z}[x] \cap \overline{\mathbb{Z}}[\delta vx],$$

where $\overline{\mathbb{Z}}$ denotes the ring of algebraic integers in \mathbb{C} . Note that the ring homomorphism $R \mapsto \mathbb{Z}$, $F \mapsto f = F(1)$ is surjective and an integer f may be represented by any F in the preimage of f under the above map. The product of two integers can

be computed by multiplying the corresponding polynomials in R and evaluating the product at 1.

Since δ is an algebraic integer, the polynomial

$$\begin{aligned} P &= 36u^4x^{12} + 36u^3x^9 + 24u^2x^6 + 6ux^3 + 1 \\ &= (\delta vx)^{12} + \delta^3(\delta vx)^9 + 4(\delta vx)^6 + \delta^3(\delta vx)^3 + 1 \end{aligned}$$

is an element of R representing the prime p .

Let $\alpha = \delta vx$. Any integer f with $\lceil -p/2 \rceil \leq f \leq \lfloor p/2 \rfloor$ can be represented by a polynomial $F \in R$ with $F(1) = f$ of the following form:

$$\begin{aligned} F &= f_0 + f_1\delta^5\alpha + f_2\delta^4\alpha^2 + f_3\delta^3\alpha^3 + f_4\delta^2\alpha^4 + f_5\delta\alpha^5 \\ &\quad + f_6\alpha^6 + f_7\delta^5\alpha^7 + f_8\delta^4\alpha^8 + f_9\delta^3\alpha^9 + f_{10}\delta^2\alpha^{10} + f_{11}\delta\alpha^{11} \\ &= f_0 + f_1 \cdot 6(vx) + f_2 \cdot 6(vx)^2 + f_3 \cdot 6(vx)^3 + f_4 \cdot 6(vx)^4 \\ &\quad + f_5 \cdot 6(vx)^5 + f_6 \cdot 6(vx)^6 + f_7 \cdot 36(vx)^7 + f_8 \cdot 36(vx)^8 \\ &\quad + f_9 \cdot 36(vx)^9 + f_{10} \cdot 36(vx)^{10} + f_{11} \cdot 36(vx)^{11}, \end{aligned}$$

where $f_1, f_2, f_3, f_4, f_5, f_7, f_8, f_9, f_{10}, f_{11} \in [-|v/2|, |v/2|)$ and $f_0, f_6 \in [-|3v|, |3v|)$. The integer f corresponds to the vector of coefficients $(f_0, f_1, \dots, f_{11})$ of F . Note that the implementation does not keep coefficients within above intervals at all times; the bounds on coefficients used in the implementation are explained in detail in Subsection 5.5.1.

5.4.2 Multiplication modulo p

Multiplication modulo p in the new representation is done in two stages, first a polynomial multiplication of the two polynomials representing the integers and second a reduction step.

Let $f, g \in \mathbb{Z}$ be two integers with corresponding polynomials $F, G \in R$ and coefficient vectors $(f_0, f_1, \dots, f_{11})$ and $(g_0, g_1, \dots, g_{11})$. The product $H = FG$ then has coefficient vector $(h_0, h_1, \dots, h_{22})$ and has the form $H = h_0 + h_1\delta^5\alpha + \dots + h_{21}\delta^3\alpha^{21} + h_{22}\delta^2\alpha^{22}$.

The reduction step uses the polynomial P to obtain a polynomial of degree at most 11. Reducing polynomials modulo P corresponds to reducing the corresponding integers modulo p . As $P = \alpha^{12} + \delta^3\alpha^9 + 4\alpha^6 + \delta^3\alpha^3 + 1$, the equation

$$\alpha^{12} = -\delta^3\alpha^9 - 4\alpha^6 - \delta^3\alpha^3 - 1$$

can be used to reduce the degree of H . The degree reduction is given in Algorithm 11.

Polynomial multiplication and degree reduction make the coefficients grow in their absolute value. Whenever the coefficients get too large they have to be reduced by carrying from one coefficient to the next. Algorithm 12 describes how this coefficient reduction is done. After the reduction, it holds that

$$h_0, h_6 \in [-|3v|, |3v|), \quad h_1, h_3, h_4, h_7, h_9, h_{10} \in [-|v/2|, |v/2|).$$

Algorithm 11 Degree reduction after polynomial multiplication**Input:** Coefficient vector $(h_0, h_1, \dots, h_{22}) \in \mathbb{Z}^{23}$ of $H \in R$ with $H(1) \equiv h \pmod{p}$.**Output:** Reduced coefficient vector $(h'_0, h'_1, \dots, h'_{11})$ of H' with $H'(1) \equiv h \pmod{p}$.

- 1: $h'_0 \leftarrow h_0 - h_{12} + 6h_{15} - 2h_{18} - 6h_{21}$
- 2: $h'_1 \leftarrow h_1 - h_{13} + h_{16} - 2h_{19} - h_{22}$
- 3: $h'_2 \leftarrow h_2 - h_{14} + h_{17} - 2h_{20}$
- 4: $h'_3 \leftarrow h_3 - h_{12} + 5h_{15} - h_{18} - 8h_{21}$
- 5: $h'_4 \leftarrow h_4 - 6h_{13} + 5h_{16} - 6h_{19} - 8h_{22}$
- 6: $h'_5 \leftarrow h_5 - 6h_{14} + 5h_{17} - 6h_{20}$
- 7: $h'_6 \leftarrow h_6 - 4h_{12} + 18h_{15} - 3h_{18} - 30h_{21}$
- 8: $h'_7 \leftarrow h_7 - 4h_{13} + 3h_{16} - 3h_{19} - 5h_{22}$
- 9: $h'_8 \leftarrow h_8 - 4h_{14} + 3h_{17} - 3h_{20}$
- 10: $h'_9 \leftarrow h_9 - h_{12} + 2h_{15} + h_{18} - 9h_{21}$
- 11: $h'_{10} \leftarrow h_{10} - 6h_{13} + 2h_{16} + 6h_{19} - 9h_{22}$
- 12: $h'_{11} \leftarrow h_{11} - 6h_{14} + 2h_{17} + 6h_{20}$
- 13: **return** $(h'_0, h'_1, \dots, h'_{11})$.

For arbitrary inputs Algorithm 11 does not guarantee any bounds on the coefficients h_2, h_5, h_8, h_{11} . For inputs with coefficients in the range that the implementation allows (see Subsection 5.5.1) the absolute value of h_2, h_5, h_8, h_{11} is only slightly larger than $v/2$ after the coefficient reduction. The function `rnd` in Algorithm 12 denotes rounding to the nearest integer.

5.5 Low-level techniques: using SIMD floating-point arithmetic

Implementations of large-integer arithmetic on 64-bit processors usually decompose a large integer into limbs of 64 bits. Arithmetic is then performed using fast 64-bit integer-multiply and -add instructions [gmp, Sha, GT07]. This implementation does not make use of these instructions but is instead based on double-precision floating-point arithmetic. Many modern processors including all processors implementing the AMD64 architecture have very fast floating-point units. This is due to the fact that the performance of many applications such as image and video processing relies on fast floating-point arithmetic rather than integer processing and that many CPU benchmarks focus on the speed of floating-point operations.

It has been shown before that one can use these fast floating-point units for high-speed cryptography and for arithmetic on large integers, see for example Bernstein in [Ber06] and [Ber04b]. In contrast to the implementation in [Ber06] which uses 80-bit floating-point values (with a 64-bit mantissa), this implementation uses 64-bit floating-point values (with a 53-bit mantissa). This makes it possible to use the single-instruction multiple-data (SIMD) instructions of the SSE2 and SSE3 instruction set operating on double-precision (64-bit) floating-point values.

Algorithm 12 Coefficient reduction

Input: Coefficient vector $(h_0, h_1, \dots, h_{11}) \in \mathbb{Z}^{12}$ of $H \in R$ with $H(1) \equiv h \pmod{p}$.

Output: Reduced coefficient vector $(h'_0, h'_1, \dots, h'_{11})$ of H' with $H'(1) \equiv h \pmod{p}$.

```

1:  $r \leftarrow \text{rnd}(h_1/v), h_1 \leftarrow h_1 - rv, h_2 \leftarrow h_2 + r$ 
2:  $r \leftarrow \text{rnd}(h_4/v), h_4 \leftarrow h_4 - rv, h_5 \leftarrow h_5 + r$ 
3:  $r \leftarrow \text{rnd}(h_7/v), h_7 \leftarrow h_7 - rv, h_8 \leftarrow h_8 + r$ 
4:  $r \leftarrow \text{rnd}(h_{10}/v), h_{10} \leftarrow h_{10} - rv, h_{11} \leftarrow h_{11} + r$ 
5:  $r \leftarrow \text{rnd}(h_2/v), h_2 \leftarrow h_2 - rv, h_3 \leftarrow h_3 + r$ 
6:  $r \leftarrow \text{rnd}(h_5/v), h_5 \leftarrow h_5 - rv, h_6 \leftarrow h_6 + r$ 
7:  $r \leftarrow \text{rnd}(h_8/v), h_8 \leftarrow h_8 - rv, h_9 \leftarrow h_9 + r$ 
8:  $r \leftarrow \text{rnd}(h_{11}/v), h_{11} \leftarrow h_{11} - rv$ 
9:  $h_9 \leftarrow h_9 - r$ 
10:  $h_6 \leftarrow h_6 - 4r$ 
11:  $h_3 \leftarrow h_3 - r$ 
12:  $h_0 \leftarrow h_0 - r$ 
13:  $r \leftarrow \text{rnd}(h_0/(6v)), h_0 \leftarrow h_0 - r \cdot 6v, h_1 \leftarrow h_1 + r$ 
14:  $r \leftarrow \text{rnd}(h_3/v), h_3 \leftarrow h_3 - rv, h_4 \leftarrow h_4 + r$ 
15:  $r \leftarrow \text{rnd}(h_6/(6v)), h_6 \leftarrow h_6 - r \cdot 6v, h_7 \leftarrow h_7 + r$ 
16:  $r \leftarrow \text{rnd}(h_9/v), h_9 \leftarrow h_9 - rv, h_{10} \leftarrow h_{10} + r$ 
17:  $r \leftarrow \text{rnd}(h_1/v), h_1 \leftarrow h_1 - rv, h_2 \leftarrow h_2 + r$ 
18:  $r \leftarrow \text{rnd}(h_4/v), h_4 \leftarrow h_4 - rv, h_5 \leftarrow h_5 + r$ 
19:  $r \leftarrow \text{rnd}(h_7/v), h_7 \leftarrow h_7 - rv, h_8 \leftarrow h_8 + r$ 
20:  $r \leftarrow \text{rnd}(h_{10}/v), h_{10} \leftarrow h_{10} - rv, h_{11} \leftarrow h_{11} + r$ 
21: return  $(h'_0, h'_1, \dots, h'_{11})$ .

```

These instructions perform two double-precision floating-point operations at once on two independent inputs laid out in 128-bit vector registers called XMM registers. The AMD64 architecture defines 16 architectural XMM registers. For example the instruction `addpd %xmm1, %xmm2` takes the low 64 bits from register `%xmm1` and the low 64 bits of register `%xmm2`, adds them as double-precision floating-point values and stores the result in the low 64 bits of register `%xmm2`; at the same time it takes the high 64 bits from register `%xmm1` and the high 64 bits of register `%xmm2`, adds them as double-precision floating-point values and stores the result in the high 64 bits of register `%xmm2`.

The most important SSE2 instructions for the implementation are the `addpd` and `mulpd` instructions. The Intel Core 2 processors (both 65-nm and 45-nm models) can dispatch up to one `mulpd` and one `addpd` instruction per cycle and thus execute 4 floating-point operations in one cycle. However, they can not execute 2 `mulpd` or 2 `addpd` instructions in the same cycle (see [Fog10a]). To arrange data in the XMM vector registers the implementation requires additional non-arithmetic instructions such as `shufpd`, `unpckhpd` and `unpcklpd`.

Note that all arithmetic instructions only have 2 operands, one of the inputs is overwritten by the output. This sometimes requires additional `mov` instructions to copy data to other registers or memory.

5.5.1 Avoiding overflows

The 64 bits of a double-precision floating-point value f are split into 1 sign bit s , 52 mantissa bits m_1, \dots, m_{52} and 11 bits for the exponent e . The real number represented by these 64 bits is

$$f = (-1)^s \cdot \left(1 + \sum_{i=1}^{52} m_i 2^{-i}\right) 2^{e-1023}.$$

The value $f = 0$ is encoded by $e = 0$; note that this constitutes an exception to the above representation. The 53-bit mantissa (52 mantissa bits and a leading implicit one bit) ensure that a double-precision floating-point value can represent integers in the interval $[-2^{53} + 1, 2^{53} - 1]$. The result of an operation on two double-precision floating-point values is guaranteed to be exact if it is an integer in this interval; values outside this interval potentially yield an overflow.

To make sure that such overflows do not occur, it is not sufficient to simply run the code on some inputs and check whether it produces the correct results; the code must not produce overflows for any valid inputs. This is ensured by the implementation of all algorithms in the C++ programming language (not using SIMD instructions) and the replacement of the `double` data type by a self-written class `CheckDouble` to represent 64-bit floating-point values. This class performs all arithmetic operations on a member variable d of type `double`. Furthermore it stores the “worst-case” absolute value in a member variable of type `uint64_t` which is updated with each operation. Before actually performing an operation it checks whether the worst-case result overflows; if it does, the program is aborted.

Updating m is straight-forward: A multiplication of (d_1, m_1) and (d_2, m_2) yields $(d_1 d_2, m_1 m_2)$, adding (d_1, m_1) and (d_2, m_2) yields $(d_1 + d_2, m_1 + m_2)$, and subtracting (d_1, m_1) from (d_2, m_2) yields $(d_2 - d_1, m_1 + m_2)$. The only divisions are by the constants v and $6v$; for those divisions it is safe to set the result to $(d/v, \lceil |m/v| \rceil)$ or $(d/6v, \lceil |m/6v| \rceil)$ respectively. The remainder of a (rounding) division by v is always between $-|v/2|$ and $|v/2|$; the maximal value of the remainder of a division by v can thus be set to $|v/2|$. Instead of using double-precision divisions the implementation uses multiplications with a double-precision approximation of $1/v$. This can cause a rounding error, therefore the maximal absolute value of the remainder is set to $\lceil (v+1)/2 \rceil$. Analogously, the maximal absolute value of the remainder of a division by $6v$ is set to $\lceil 3v \rceil + 1$.

For all constants involved in the pairing computation the maximal value is initialized with the actual value. The inputs to the pairing are initialized as worst-case output of the reduction described in Algorithm 12.

The high-performance version of the software does not use the `CheckDouble` class but the `double` data type. All speed-critical functions in this version are implemented in `qasm` (see Section 2.8) using SIMD instructions where possible. The operations on double-precision floating-point values in this `qasm` implementation are the same as in the C++ version for which the `CheckDouble` class automatically verified that no overflows can occur. The resulting software has passed a bilinearity

and non-degeneracy test on 1,000,000 random inputs, each test involving 3 pairing computations.

5.5.2 Implementation of field arithmetic

The 12 coefficients f_0, \dots, f_{11} of a polynomial F representing an element $f \in \mathbb{F}_p$ (see Section 5.4) are stored consecutively in a double array of size 12. The 24 coefficients g_0, \dots, g_{11} and h_0, \dots, h_{11} representing an element $(gi + h) \in \mathbb{F}_{p^2} = \mathbb{F}_p(i)$ are stored interleaved in a double array of size 24 as $(h_0, g_0 | h_1, g_1 | \dots | h_{11}, g_{11})$. In the following descriptions, all SIMD instructions operate on every two adjacent double values of this representation. Observe that the implementations do not minimize the number of *instructions* but try to minimize the number of *cycles*.

$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication. Multiplication of $ai + b$ and $ci + d$, laid out in memory as $op_1 = (b_0, a_0 | \dots | b_{11}, a_{11})$ and $op_2 = (d_0, c_0 | \dots | d_{11}, c_{11})$, is done as follows: First swap the coefficients in op_2 to obtain

$$t_1 = (c_0, d_0 | c_1, d_1 | \dots | c_{11}, d_{11}).$$

Then duplicate once the lower and once the upper coefficients of op_1 to obtain

$$t_2 = (a_0, a_0 | a_1, a_1 | \dots | a_{11}, a_{11})$$

and

$$t_3 = (b_0, b_0 | b_1, b_1 | \dots | b_{11}, b_{11}).$$

Multiply the low double values of t_2 by $i^2 = -1$ and obtain

$$t_2 = (-a_0, a_0 | -a_1, a_1 | \dots | -a_{11}, a_{11})$$

Then perform digitsliced multiplication of t_1 with t_2 and of t_3 with op_2 to obtain

$$r_0 = (-(ac)_0, (ad)_0 | \dots | -(ac)_{22}, (ad)_{22})$$

and

$$r_1 = ((bd)_0, (bc)_0 | \dots | (bd)_{22}, (bc)_{22}).$$

Finally add r_0 to r_1 (merged with the digitsliced multiplications) and apply polynomial reduction (Algorithm 11) and coefficient reduction (Algorithm 12) to obtain interleaved coefficients of $(ai + b)(ci + d) = ((ad + bc)i + (bd - ac))$.

The parallel digitsliced multiplication uses the schoolbook algorithm resulting in 144 multiplications (`mulpd`), 121 additions (`addpd`), and 10 more multiplications by 6 (`mulpd`). Experiments with Karatsuba multiplication did not show any performance gain.

Computing the rounded quotient and the remainder in the coefficient reduction could be done using multiplication by a double-precision approximation to $1/v$, using the `roundpd` instruction on the result, multiplying by v and subtracting the result from the original value to obtain the remainder. The `roundpd` instruction is part of

the SSE4.1 instruction set which is not available on 65 nm Core 2 processors and AMD processors. Therefore rounding is implemented as addition and subsequent subtraction of a constant as explained for example by Bernstein in [Ber04b, Theorem 2.4].

\mathbb{F}_{p^2} **squaring**. Squaring an element $ai + b \in \mathbb{F}_{p^2}$, laid out in memory as $op_1 = (b_0, a_0 | \dots | b_{11}, a_{11})$ is done as follows:

First swap the coefficients to obtain

$$t_1 = (a_0, b_0 | \dots | a_{11}, b_{11}).$$

then use the `subsd` instruction on op_1 and t_1 to obtain

$$op_1 = (b_0 - a_0, a_0 | \dots | b_{11} - a_{11}, a_{11}).$$

Then copy t_1 and duplicate the high double values to obtain

$$t_2 = (b_0, b_0 | \dots | b_{11}, b_{11}).$$

Adding t_1 and t_2 yields

$$t_2 = (b_0 + a_0, 2b_0 | \dots | b_{11} + a_{11}, 2b_{11}).$$

Now use digit-sliced multiplication on op_1 and t_2 to obtain

$$r = (((b - a)(a + b))_0, (2ab)_0 | \dots | ((b - a)(a + b))_{22}, (2ab)_{22}).$$

Applying polynomial reduction and coefficient reduction yields the coefficients of the result $(2ab)i + (b^2 - a^2)$.

$\mathbb{F}_{p^2} \times \mathbb{F}_p$ **multiplication**. Evaluating the line functions requires multiplications of an element of \mathbb{F}_{p^2} with an element of \mathbb{F}_p . This is implemented using the same parallel schoolbook multiplication which is used in the $\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication. This requires first duplicating the coefficients of the \mathbb{F}_p element in memory.

\mathbb{F}_{p^2} **short coefficient reduction**. Additions, subtractions and multiplications with small constants in \mathbb{F}_{p^2} can all be implemented using 12 SIMD instructions. They produce results which may have coefficients that are too large to go as input into a multiplication or squaring but are still so small that they do not require the full-fledged coefficient reduction from Algorithm 12. The coefficients of such inputs to a multiplication or squaring are reduced in a short coefficient reduction as follows: first carry from coefficient f_{11} to f_0, f_3, f_6 and f_9 . Then carry from all odd coefficients f_1, f_3, \dots, f_9 and then from all even coefficients f_0, f_2, \dots, f_{10} .

$\mathbb{F}_{p^{12}}$ **inversion**. The final exponentiation involves one inversion in $\mathbb{F}_{p^{12}}$. This can be computed with only one inversion in \mathbb{F}_p and several multiplications as described in, e.g., [HMS08, Sec. 2]. Inversion in \mathbb{F}_p is performed as exponentiation with $p - 2$ using a simple square-and-multiply algorithm. There certainly exist faster methods to compute inverses in \mathbb{F}_p , but this is the easiest way to ensure constant-time behavior of the inversion and the single inversion in $\mathbb{F}_{p^{12}}$ accounts for less than 4 percent of the total computation time.

5.6 Performance results and comparison

This section gives benchmarking results of the most important operations used in the pairing on different microarchitectures. The implementation is optimized for the 65-nm and 45-nm Intel Core 2 processors, but the benchmark results also include timings from a 32-nm Intel Xeon and an AMD K10 processor showing that the software is also fast on those microarchitectures. For details of the machines used for benchmarking see Table 5.1. The cycle counts for each function are obtained as follows: Read the cycle counter, perform 50 calls to the function to benchmark, and read the cycle counter again. Compute the average of cycles taken by these 50 calls. Repeat that 20 times and print all 20 averages. Then run the same benchmarking program again, repeat this 50 times, this generates a total of 1000 cycle counts for each function. The reason for measuring in separate runs of the same program is that some variation in the timings only occurs in separate runs. This kind of variation is most likely due to different caching behavior for different mappings of virtual to physical memory. Tables 5.2, 5.3, 5.4 and 5.5 report the median and quartiles of these numbers.

5.6.1 Comparison with previous work

There exist several descriptions and benchmarks of software implementations of cryptographic pairings. Implementations targeting the 128-bit security level usually use 256-bit BN curves.

The software presented in [HMS08] takes 10,000,000 cycles to compute the R-ate pairing over a 256-bit BN curve on one core of an Intel Core 2 processor; the same computation on one core of an AMD Opteron processor also takes 10,000,000 cycles. Unpublished benchmarks of a newer version of that software (included in the Miracl library [Sha]) are claimed to take 7,850,000 cycles on an Intel Core 2 Duo T5500 processor [Sco10]. The software presented in [NBS08] takes 29,650,000 cycles to compute the ate pairing over a 256-bit BN curve on one core of a Core 2 Duo processor. Software presented in [GGP09] takes 23,319,673 cycles to compute the ate pairing over a 256-bit BN curve on one core of an Intel Core 2 Duo processor; another implementation described in the same paper takes 14,429,439 cycles to compute the ate pairing on two cores of an Intel Core 2 Duo processor.

The software presented in [ALH10] also targets the 128-bit security level and computes the η_T pairing on a supersingular curve over a binary field in 17,400,000 cycles on one core of a 45 nm Intel Core 2 processor. The paper also presents performance numbers of the same pairing computation on multiple cores: for example the computation on 4 cores of a 45 nm Intel Core 2 processor takes 5,080,000 cycles, and on 8 cores it takes 3,020,000 cycles.

The software presented here computes the optimal ate pairing in 4,142,290 cycles on one core of an Intel Core 2 Quad Q6600 processor and is thus more than twice as fast as the fastest previously published result and more than 1.8 times faster than previous unpublished results.

Table 5.1: Machines used for benchmarking the pairing implementation

latour		
Affiliation	Eindhoven University of Technology	
CPU	Intel Core 2 Quad Q6600, 2394 MHz	
Compilers	gcc	gcc (Ubuntu 4.3.3-5ubuntu4) 4.3.3
	gcc-4.1	gcc-4.1 (GCC) 4.1.3 20080623 (prerelease) (Ubuntu 4.1.2-24ubuntu1)
	gcc-4.2	gcc-4.2 (GCC) 4.2.4 (Ubuntu 4.2.4-5ubuntu1)
	g++	g++ (Ubuntu 4.3.3-5ubuntu4) 4.3.3
berlekamp		
Affiliation	National Taiwan University	
CPU	Intel Core 2 Quad Q9550, 2833 MHz	
Compilers	gcc	gcc (GCC) 4.4.1 20090725 (Red Hat 4.4.1-2)
	gcc34	gcc34 (GCC) 3.4.6 20060404 (Red Hat 3.4.6-13)
	g++	g++ (GCC) 4.4.1 20090725 (Red Hat 4.4.1-2)
dragon3		
Affiliation	Academia Sinica	
CPU	Intel Xeon X5650, 2668 MHz	
Compilers	gcc	gcc (GCC) 4.4.4 20100630 (Red Hat 4.4.4-10)
	g++	g++ (GCC) 4.4.4 20100630 (Red Hat 4.4.4-10)
chukonu		
Affiliation	University of Illinois at Chicago	
CPU	AMD Phenom X4 955, 3210.623 MHz	
Compilers	gcc	gcc (Ubuntu 4.4.1-4ubuntu9) 4.4.1
	gcc-4.3	gcc-4.3 (Ubuntu 4.3.4-5ubuntu1) 4.3.4

Table 5.2: Cycle counts of various operations involved in the pairing computation on `latour`, all code compiled with `gcc -std=c99 -O3 -fomit-frame-pointer`

	25% quartile	median	75% quartile
$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication	589.32	590.94	599.40
\mathbb{F}_{p^2} squaring	356.58	357.30	360.18
$\mathbb{F}_{p^2} \times \mathbb{F}_p$ multiplication	371.70	371.70	384.30
\mathbb{F}_{p^2} short coeff. reduction	73.80	73.80	73.98
line-function eval. (addition)	8373.51	8388.99	8417.70
line-function eval. (doubling)	6501.24	6518.16	6569.28
$\mathbb{F}_{p^{12}}$ multiplication	14726.52	14783.67	14883.75
$\mathbb{F}_{p^{12}}$ squaring	10452.6	10514.70	10571.94
$\mathbb{F}_{p^{12}}$ line-function mult.	10016.64	10042.02	10117.71
$\mathbb{F}_{p^{12}}$ inversion	158006.25	158226.39	160530.21
Miller loop	2122417.26	2129149.35	2135567.07
optimal ate pairing	4135767.30	4142289.69	4153829.22

Table 5.3: Cycle counts of various operations involved in the pairing computation on `berlekamp`, all code compiled with `gcc -std=c99 -O3 -fomit-frame-pointer`

	25% quartile	median	75% quartile
$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication	588.72	591.10	602.22
\mathbb{F}_{p^2} squaring	356.32	357.00	358.88
$\mathbb{F}_{p^2} \times \mathbb{F}_p$ multiplication	370.94	371.12	377.06
\mathbb{F}_{p^2} short coeff. reduction	73.78	73.94	73.96
line-function eval. (addition)	8345.64	8393.42	8451.05
line-function eval. (doubling)	6486.94	6520.86	6554.68
$\mathbb{F}_{p^{12}}$ multiplication	14691.57	14756.25	14836.07
$\mathbb{F}_{p^{12}}$ squaring	10463.34	10514.16	10582.76
$\mathbb{F}_{p^{12}}$ line-function mult.	10015.2	10058.30	10137.18
$\mathbb{F}_{p^{12}}$ inversion	159127.14	159291.11	159499.18
Miller loop	2128363.2	2134868.33	2139546.31
optimal ate pairing	4147904.63	4161923.16	4171976.98

Table 5.4: Cycle counts of various operations involved in the pairing computation on dragon3, all code compiled with gcc -std=c99 -O3 -fomit-frame-pointer

	25% quartile	median	75% quartile
$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication	459.54	460.20	480.18
\mathbb{F}_{p^2} squaring	333.22	333.44	334.94
$\mathbb{F}_{p^2} \times \mathbb{F}_p$ multiplication	338.44	338.60	352.18
\mathbb{F}_{p^2} short coeff. reduction	63.76	63.82	64.66
line-function eval. (addition)	6976.15	7032.39	7205.66
line-function eval. (doubling)	5691.47	5754.10	5928.12
$\mathbb{F}_{p^{12}}$ multiplication	11792.88	11883.96	12107.64
$\mathbb{F}_{p^{12}}$ squaring	8383.82	8504.45	8755.54
$\mathbb{F}_{p^{12}}$ line-function mult.	7962.10	8098.65	8321.54
$\mathbb{F}_{p^{12}}$ inversion	143130.64	143181.8	143294.32
Miller loop	1750044.15	1756319.76	1764711.44
optimal ate pairing	3451373.59	3458785.04	3465786.78

Table 5.5: Cycle counts of various operations involved in the pairing computation on chukonu, all code compiled with gcc -std=c99 -O3 -fomit-frame-pointer

	25% quartile	median	75% quartile
$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication	585.74	587.74	589.04
\mathbb{F}_{p^2} squaring	364.64	365.60	366.56
$\mathbb{F}_{p^2} \times \mathbb{F}_p$ multiplication	372.32	372.32	373.38
\mathbb{F}_{p^2} short coeff. reduction	108.42	108.42	108.42
line-function eval. (addition)	8814.76	8830.56	8869.52
line-function eval. (doubling)	6940.60	6958.66	7002.02
$\mathbb{F}_{p^{12}}$ multiplication	15958.68	15988.69	16028.41
$\mathbb{F}_{p^{12}}$ squaring	11416.48	11427.58	11456.64
$\mathbb{F}_{p^{12}}$ line-function mult.	10583.80	10612.32	10638.33
$\mathbb{F}_{p^{12}}$ inversion	158166.11	158605.44	159133.54
Miller loop	2262322.83	2266533.21	2268562.22
optimal ate pairing	4460435.73	4468240.38	4476366.16

Table 5.6: Cycle counts of the implementation presented in [BGM⁺10]. Curve parameters: $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$, with $u = 2^{62} - 2^{54} + 2^{44}$, BN curve: $y^2 = x^3 + 5$ over \mathbb{F}_p , all code compiled with `g++ -x c++ -O3 -fomit-frame-pointer -msse2 -mfpmath=sse`

	latour	berlekamp
$\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ multiplication	595.26	583.44
\mathbb{F}_{p^2} squaring	484.92	484.16
optimal ate pairing	3145815.90	3141289.08

5.6.2 Comparison with [BGM⁺10]

After the results of this chapter were presented in [NNS10], Beuchat et al. reported new speed records set by an implementation of the optimal ate pairing over a 254-bit Barreto-Naehrig curve: 2,490,000 cycles on an Intel Core i7 860 processor and 3,140,000 cycles on an Intel Core 2 T7100 processor [BGM⁺10]. This implementation uses Montgomery representation [Mon85] for arithmetic in \mathbb{F}_p which does not exploit the special structure of p . Multiplication in \mathbb{F}_p is based on the `mul` instruction which is capable of multiplying two 64-bit unsigned integers to produce a 128-bit result every 4 cycles on Core 2 processors and every 2 cycles on the newer Core i7 processors [Fog10a]. The performance of this integer-multiplication instruction on the Core i7 processor is the reason why the implementation clearly outperforms the implementation presented here on this microarchitecture. However, also on the Core 2 processors, the primary target of the implementation described here, the optimal ate pairing is computed faster by the software presented in [BGM⁺10]. For a direct comparison Table 5.6 gives benchmark results of pairing computation and \mathbb{F}_{p^2} -arithmetic operations from the implementation presented in [BGM⁺10]. These results were measured on the same two Intel Core 2 machines and using the same median of averages as the benchmarks in Tables 5.2 to 5.5. There are three main reasons why the followup implementation in [BGM⁺10] achieves even higher speeds than the software described in this chapter.

Restricted choice of a BN curve. As described in Section 5.4, the primary target was to make multiplication and squaring in \mathbb{F}_{p^2} as fast as possible. The representation of integers described in Section 5.4 requires the BN parameter u to be a third power which clearly restricts the choice of a curve. The curve used in [BGM⁺10] uses the parameter $u = 2^{62} - 2^{54} + 2^{44}$; with this parameter the Miller loop of the optimal ate pairing contains only 6 addition steps and each exponentiation with u in the final exponentiation involves only 2 multiplications in $\mathbb{F}_{p^{12}}$. Our implementation uses 12 more addition steps in the Miller loop, each taking about $8389 + 10042 = 18431$ cycles (line-function evaluation and $\mathbb{F}_{p^{12}}$ line-function multiplication, cycle-count medians of `latour`). Each of the three exponentiations with u involves 10 more multiplications in $\mathbb{F}_{p^{12}}$, each taking about 14784 cycles. The overhead from the

restricted choice of curve thus accounts for a total of $12 \cdot 18431 + 30 \cdot 14784 = 664692$ cycles.

Short coefficient reductions. Squaring, addition, and subtraction in \mathbb{F}_{p^2} are faster in the implementation described here, multiplication achieves similar speeds. However, the redundant representation of field elements incurs additional costs. As explained in Section 5.5 before some multiplications the coefficients of outputs of previous additions have to be reduced. A total of 5556 such short coefficient reductions are carried out during the computation of the optimal ate pairing, each taking about 74 cycles. The overhead from these short coefficient reductions is thus about 411144 cycles.

SIMD overhead. As explained in Section 5.5, the Intel Core 2 processors can execute 2 double-precision floating-point multiplications and 2 double-precision floating-point additions in one cycle. As the corresponding `mulpd` and `addpd` instruction are vector instructions, additional shuffle instructions are required to arrange inputs in memory and registers as required. These shuffles account for an overhead for about 57 cycles in the \mathbb{F}_{p^2} multiplication and are one reason why multiplication in \mathbb{F}_{p^2} is not considerably faster than the Montgomery multiplication in [BGM⁺10]. The other reason why multiplication is not faster is the fact that the vector instructions require an even number of \mathbb{F}_p multiplications, multiplication in \mathbb{F}_{p^2} can thus not benefit from the Karatsuba technique which is used in [BGM⁺10]. Note that squaring in \mathbb{F}_{p^2} , which can use the same technique as [BGM⁺10] and which has smaller overhead from shuffle instructions (only about 41 cycles) is using only 357 cycles and is considerably faster than squaring in \mathbb{F}_{p^2} as implemented in [BGM⁺10] which requires 485 cycles.

The software presented in this chapter is not the fastest pairing implementation for Intel and AMD processors, but it is the fastest pairing software based on double-precision floating-point arithmetic. For architectures and microarchitectures with a stronger focus on efficient double-precision floating-point performance (compared to integer performance), the representation of elements of \mathbb{F}_p with the special reduction for BN primes will most likely yield the best performance results, in particular if high-performance double-precision floating-point instructions can be used without overhead from SIMD requirements.

6

Solving the elliptic-curve discrete-logarithm challenge ECC2K-130

The security of elliptic-curve cryptography relies on the hardness of the elliptic-curve discrete-logarithm problem (ECDLP): Given an elliptic curve E over a finite field \mathbb{F}_q and two points $P \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle$, find an integer k such that $Q = [k]P$. Here $[k]$ denotes scalar multiplication with k . If the order of $\langle P \rangle$ is prime, the best algorithm known to solve this problem (for most elliptic curves) is Pollard’s rho algorithm [Pol78], which solves the problem in time $O(\sqrt{|\langle P \rangle|})$.

This complexity of Pollard’s rho algorithm is the reason that theoretical considerations of the hardness of the ECDLP state that the ECDLP can be solved in $O(\sqrt{n})$, where n is the group order (see e.g. the documents on [Gir08]). How hard the ECDLP is in practice requires more investigation of the constant factors hidden in $O(\sqrt{n})$, optimization techniques of Pollard’s rho algorithm and performance measurements of a thoroughly optimized implementation of the algorithm on real state-of-the-art hardware. In order to “increase the cryptographic community’s understanding and appreciation of the difficulty of the ECDLP” Certicom in 1997 issued several ECDLP challenges [Cer97a].

This chapter describes an attack against one of these challenges, namely challenge ECC2K-130. Sections 6.1 and 6.2 explain how exactly Pollard’s rho algorithm is used to solve the specific given ECDLP. Sections 6.3 and 6.4 describe implementations of the attack on two different architectures: the Cell Broadband Engine and NVIDIA graphics processing units (GPUs). Section 6.5 discusses the performance achieved by the two implementations.

This chapter is based on joint work with Bos, Kleinjung, and Niederhagen published in [BKNS10], and joint work with Bernstein, Chen, Cheng, Lange, Niederha-

gen and Yang published in [BCC⁺10]. Aside from minor changes to phrasing and notation the main differences between this chapter and [BKNS10], and [BCC⁺10] are the following:

- the background given in Sections 6.1 and 6.2 is merged from the respective sections of [BKNS10] and [BCC⁺10], and also contains material from [BBB⁺09b] and [BBB⁺09a];
- Section 6.3 describes only the fastest implementation of the iteration function, [BKNS10] contains more details on the design decisions and additionally describes two other, slower, implementations;
- the implementation for Cell processors described in Section 6.3 contains some new optimizations and is thus faster than the one presented in [BKNS10];
- Section 6.5 discusses the performance numbers on the two different architectures, this discussion is not contained in [BKNS10] or [BCC⁺10].

6.1 The parallel version of Pollard's rho algorithm

To solve large instances of the ECDLP one usually uses a parallelized version of Pollard's rho algorithm due to van Oorschot and Wiener [vOW99]. Several variants of this parallelized version have been proposed and used to solve instances of the ECDLP see for example [Har00] and [BKM09]. For the attack against ECC2K-130 the following parallel version of Pollard's rho algorithm is used; for credit see [BBB⁺09b].

The algorithm is based on a client-server approach in which each client does the following:

1. From a random seed s , generate a pseudo-random starting point R_0 as a known linear combination in P and Q : $R_0 = a_0P + b_0Q$;
2. apply a pseudo-random iteration function f to obtain a sequence $R_{i+1} = f(R_i)$, where f must preserve knowledge about the linear combination in P and Q ;
3. after each step of the iterative applications of f , check whether the output point belongs to an easy-to-recognize subset of $\langle P \rangle$ of so-called *distinguished points*;
4. if at some point a distinguished point D is reached, send (D, s) to the server and go to step 1.

The server receives all the incoming pairs (D_i, s_i) and does the following:

1. Search the pairs for a *collision*, i.e., two pairs $(D_i, s_i), (D_j, s_j)$ with $D_i = D_j$ and $s_i \neq s_j$;
2. when a collision is found, compute the linear combinations in P and Q for $D_i = a_iP + b_iQ$ and $D_j = a_jP + b_jQ$;

3. obtain the discrete logarithm of Q to the base P as $k = \frac{a_j - a_i}{b_i - b_j} \pmod{\text{ord}(P)}$, if $(b_i - b_j) \not\equiv 0 \pmod{\text{ord}(P)}$.

The expected number of calls to the iteration function f of this parallel version of Pollard's rho algorithm is $\sqrt{\frac{\pi(\langle P \rangle)}{2}}$ assuming perfectly random behavior of the iteration function f [Har60].

6.2 ECC2K-130 and choice of the iteration function

The specific ECDLP addressed here is given in the Certicom challenge list [Cer97a] as challenge ECC2K-130. The given elliptic curve is a Koblitz curve $E : y^2 + xy = x^3 + 1$ over the finite field $\mathbb{F}_{2^{131}}$; the two given points P and Q have order l , where l is a 129-bit prime. The challenge is to find an integer k such that $Q = [k]P$. This section describes the choice of the iteration function and the distinguished-point property used in the implementations to solve this specific ECDLP. A comparison to other possible choices is given in [BBB⁺09a].

Given a point (x, y) on the curve it is computationally easy to determine its negative $(x, x + y)$. A common optimization technique for Pollard's rho algorithm is to work on equivalence classes modulo negation of points. This reduces the expected number of iterations by a factor of up to $\sqrt{2}$ [WZ98, GLV00], but it requires the iteration function to be well-defined on equivalence classes modulo negation.

For Koblitz curves a generalization of this idea can be used to reduce the expected number of iterations even further: As the curve equation has coefficients in \mathbb{F}_2 , the i -th powers of the Frobenius endomorphism

$$\phi^i : E(\mathbb{F}_{2^{131}}) \rightarrow E(\mathbb{F}_{2^{131}}), (x, y) \mapsto (x^{2^i}, y^{2^i}), i = 1, \dots, 130$$

are efficiently computable automorphisms besides negation. The number of iterations can thus be reduced by a total factor of up to $\sqrt{2 \cdot 131}$ by considering orbits under the 131 i -th power Frobenius endomorphisms and negation.

The attack described in this chapter uses the following definitions of distinguished points and iteration function: A point R_i (representing an equivalence class) is defined to be *distinguished* if the Hamming weight of the x -coordinate in normal-basis representation $\text{HW}(x_{R_i})$ is smaller than or equal to 34. Observe that this property respects the orbits of negation (because the x -coordinate remains the same) and of the Frobenius endomorphism (because squaring in normal basis is just a cyclic rotation of bits, not affecting the Hamming weight). The iteration function is defined as

$$R_{i+1} = f(R_i) = \phi^j(R_i) + R_i,$$

where $j = ((\text{HW}(x_{R_i})/2) \pmod{8}) + 3$. The iteration function is well-defined on equivalence classes modulo negation and application of ϕ , because both are endomorphisms.

The restriction of ϕ to $\langle P \rangle$ corresponds to scalar multiplication with some scalar r . For an input $R_i = a_i P + b_i Q$ the output of f will be $R_{i+1} = (r^j a_i + a_i) P + (r^j b_i + b_i) Q$.

When a collision has been detected, it is possible to recompute the two corresponding iterations and update the coefficients a_i and b_i following this rule. This gives the coefficients to compute the discrete logarithm.

6.2.1 Computing the iteration function

Computing the iteration function requires one application of ϕ^j and one elliptic-curve addition. Furthermore it requires conversion of the x -coordinate of the resulting point to normal basis, if a polynomial-basis representation is used, and a check whether the resulting point is distinguished.

Many applications use so-called inversion-free coordinate systems to represent points on elliptic curves (see, e.g., [HMV04, Sec. 3.2]) to speed up the computation of point multiplications. These coordinate systems use a redundant representation for points. Identifying distinguished points requires a unique representation, this is why points have to be represented in affine coordinates. Elliptic-curve addition in affine Weierstrass coordinates on the given elliptic curve requires 2 multiplications, one squaring, 6 additions, and 1 inversion in $\mathbb{F}_{2^{131}}$ (see, e.g. [BLb]). Application of ϕ^j means computing the 2^j -th powers of the x - and the y -coordinate. In total, one iteration takes 2 multiplications, 1 squaring, 2 computations of the form r^{2^m} , with $3 \leq m \leq 10$, 1 inversion, 1 conversion to normal-basis, and one Hamming-weight computation. Computations of the form r^{2^m} will be referred to as *m-squaring* in the following.

A note on the inversion. The relatively costly inversion can be sped up by batching several inversions and using Montgomery's trick [Mon87]: m batched inversions can be computed with $3(m - 1)$ multiplications and one inversion. For example, $m = 64$ batched elliptic curve additions take $2 \cdot 64 + 3 \cdot (64 - 1) = 317$ multiplications, 64 squarings and 1 inversion. This corresponds to 4.953 multiplications, 1 squaring and 0.016 inversions for a single elliptic-curve addition.

6.2.2 Bitsliced binary-field arithmetic

Most modern processors, including the architectures considered in this chapter, support fast arithmetic on integers and floating-point values but not on binary polynomials. Therefore, binary-field arithmetic, which cannot benefit from fast integer or floating-point arithmetic instructions, was commonly believed to be significantly less efficient in software than prime-field arithmetic which naturally makes use of integer or floating-point arithmetic.

Bernstein in [Ber09a] showed that binary-field arithmetic can be faster than prime-field arithmetic also in software when considering many independent, batched operations. The idea behind his implementation is applying the bitslicing technique to finite fields. As for the case of AES described in Section 3.4 of Chapter 3 the idea is to simulate a hardware implementation with bit-logical instructions.

Bitslicing elements of $\mathbb{F}_{2^{131}}$ means storing the 131 coefficients in 131 registers, one register per coefficient. Implementations on architectures with fewer registers

have to use spills to memory to compensate for the lack of registers. All algorithms need to operate on $k \cdot n$ inputs in parallel where n is the width of the registers, to make best use of the computational power of the bit-logical instructions. The value k is typically 1 for straightforward bitsliced implementations; Section 6.4 describes that the implementation on GPUs needs a higher value ($k = 4$) to achieve best performance. Note that bitsliced implementations have to work on significantly more data: all inputs, outputs and buffers need to have $k \cdot n$ times the size of a non-bitsliced implementation.

6.2.3 Representing elements of $\mathbb{F}_{2^{131}}$

An important choice in the implementation of binary-field arithmetic is the choice of the basis used to represent the elements as vectors over \mathbb{F}_2 . Polynomial bases are of the form $(1, z, z^2, z^3, \dots, z^{130})$, so the basis elements are increasing powers of some element $z \in \mathbb{F}_{2^{131}}$. Normal bases are of the form $(\alpha, \alpha^2, \alpha^4, \dots, \alpha^{2^{130}})$, so each basis element is the square of the previous one.

There are two advantages of using normal-basis representation. First, the iteration function needs the normal-basis representation of the x -coordinate to determine the exponent of the Frobenius endomorphism, and to check whether the point is distinguished. Using this representation also for all computations thus saves one basis conversion. Second, all squarings and m -squarings in normal-basis representation correspond to very fast cyclic shifts.

The disadvantage of using normal-basis representation for odd-degree extensions of \mathbb{F}_2 is performance of multiplications that were long believed to be much slower than multiplications in polynomial basis (see, e.g., [HMOV04, Section 2.5]). However in [Sho07] Shokrollahi proposed a method for efficient normal-basis multiplication in binary fields with an optimal normal basis (see also [vzGSS07]). For the finite field $\mathbb{F}_{2^{131}}$ an optimal normal basis exists [MOVW89] and bitsliced implementations of the ECC2K-130 iteration function using normal-basis representation of elements of $\mathbb{F}_{2^{131}}$ and the Shokrollahi multiplier are faster than implementations based on polynomial-basis representation. The following paragraphs first describe the idea of the Shokrollahi multiplier including improvements by Bernstein and Lange described in [BBB⁺09b] and then give a summary of the notion of “optimal polynomial bases” introduced by Bernstein and Lange in [BL10].

The Shokrollahi multiplier. An element of $\mathbb{F}_{2^{131}}$ in type-2 optimal-normal-basis representation is of the form

$$f_0(\zeta + \zeta^{-1}) + f_1(\zeta^2 + \zeta^{-2}) + f_2(\zeta^4 + \zeta^{-4}) + \dots + f_{130}(\zeta^{2^{130}} + \zeta^{-2^{130}}),$$

where ζ is a 263rd root of unity in $\mathbb{F}_{2^{131}}$. The coefficients of both inputs to a multiplication are first permuted into coefficients of

$$\begin{aligned} & (\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^{2^{83}} + \zeta^{-2^{83}}, \zeta^4 + \zeta^{-4}, \zeta^{2^{20}} + \zeta^{-2^{20}}, \dots, \zeta^{2^{130}} + \zeta^{-2^{130}}) \\ & = (\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \zeta^4 + \zeta^{-4}, \zeta^5 + \zeta^{-5}, \dots, \zeta^{131} + \zeta^{-131}); \end{aligned}$$

in the following this representation will be called *permuted-normal-basis representation*. The inputs are then further transformed to obtain coefficients in the polynomial basis

$$\zeta + \zeta^{-1}, (\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \dots, (\zeta + \zeta^{-1})^{131}.$$

Now polynomial multiplication is straightforward and yields coefficients of

$$(\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \dots, (\zeta + \zeta^{-1})^{262}.$$

This representation is transformed to obtain coefficients of

$$\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \dots, \zeta^{262} + \zeta^{-262}.$$

Conversion to permuted normal basis just requires adding appropriate coefficients; for example ζ^{200} is the same as ζ^{-63} (because ζ is a 263rd root of unity) and thus $\zeta^{200} + \zeta^{-200}$ is the same as $\zeta^{63} + \zeta^{-63}$. The inverse of the initial permutation then yields the result of the multiplication in normal-basis representation.

Optimal polynomial bases. In the iteration function some outputs of a multiplication are used again as input to another multiplication. In such cases Bernstein and Lange avoid some of the bit operations of the double-length conversion by keeping the lower 131 coefficients in polynomial-basis representation and using the conversion routine only to compute the polynomial reduction, ending up in polynomial-basis representation [BL10]. They define an *optimal polynomial basis* to mean the basis used on the polynomial-representation side and propose this representation as an alternative to normal-basis representation for applications that need many multiplications and only occasional squarings. The GPU implementation includes these improvements and uses two different multiplication routines: Both multiplication routines accept input in optimal-polynomial-basis representation, a *ppp multiplication* delivers output in optimal-polynomial-basis representation, and a *ppn multiplication* delivers output in permuted-normal-basis representation. For details of optimal polynomial bases see [BL10].

6.3 Implementing the iteration function on Cell processors

The implementation of the ECC2K-130 iteration function for Cell processors has been presented in [BKNS10] before Bernstein and Lange introduced optimal polynomial bases in [BL10]. The implementation uses an optimal-normal-basis representation for all field operations. Multiplication is done using the Shokrollahi multiplier described in Subsection 6.2.3.

As already explained in Section 4.2 each Synergistic Processor Unit of the Cell Broadband Engine can dispatch one bit-logical instruction on 128-bit registers per cycle. Additionally a load or store instruction can be dispatched in the same cycle. A natural lower bound on the number of cycles per Pollard-rho iteration thus comes from the number of bit-logical instructions per iteration. The implementation of the

ECC2K-130 iteration function is not able to meet this lower bound. This is due to the fact that it has to deal with a significantly larger amount of data than in the case of bitsliced AES described in Section 3.4 of Chapter 3 and some algorithms are simply bottlenecked by loads and stores. Furthermore, all code has to fit into only 256 KB of local storage alongside the active set of data. A fully unrolled iteration function on 128 bitsliced inputs in parallel would not fit into the local storage. Organizing code in several functions incurs call overhead.

The following description of the implementation first considers in detail how the field operations are implemented and then discusses the overall performance of the iteration function. As described above, all field operations operate on elements in bitsliced normal-basis representation, and all cycle counts are given for operations on a batch of 128 parallel operations.

Most cycle counts are measured using the integrated decremter of the SPU (see [IBM08, Sec. 13.3.3]). Cycle counts for small portions of code which are not implemented as a separate function are obtained using the `spu_timing` tool included in the IBM SDK for Multicore Acceleration [IBM], which performs static analysis of the code and gives accurate cycle counts for most code. For code containing a load or store instruction in almost every cycle the cycle counts reported by `spu_timing` can be overly optimistic. For more details see [BKNS10, Sections 2.3 and 2.4]. All cycle counts are given with information about whether they were obtained using `spu_timing`, or measured by running the code.

Multiplication. The normal-basis multiplication uses the conversion to polynomial basis as described in Subsection 6.2.3. Both conversion of inputs to polynomial basis and conversion of the result to normal basis (including reduction) use fully unrolled assembly functions, which are scheduled automatically by self-written scheduler scripts. One input conversion takes 434 cycles (measured). Output conversion including reduction takes 1288 cycles (measured).

The main part of the multiplication routine is still efficient 131-coefficient binary-polynomial multiplication. The smallest known number of bit operations required to multiply two 131-coefficient polynomials over \mathbb{F}_2 is 11961 [Ber09a, Section 2], see also [Ber09b]. However, converting the sequence of bit operations from [Ber09b] to C syntax and feeding it to `spu-gcc` (version 4.1.1) does not compile because the size of the resulting function exceeds the size of the local storage. After reducing the number of variables for intermediate results and some more tweaks the compiler produces functioning code, which has a code size of more than 100 KB and requires more than 20000 cycles to compute a multiplication.

A more compact multiplication routine with better scheduling sacrifices some bit operations and composes the 131-coefficient multiplication of 9 33-coefficient multiplications using two levels of the Karatsuba multiplication technique [KO63]. One of these multiplications is actually only a 32-coefficient multiplication; in order to keep code size small it is performed as a 33-coefficient multiplication with leading coefficient zero. The 9 65-coefficient results are combined using the refined-Karatsuba techniques described in [Ber09a].

The smallest known number of bit operations for 33-coefficient binary-polynomial multiplication is 1286 [Ber09b]. A self-written scheduler for the bit operation sequence from [Ber09b] generates code that takes 1303 cycles (`spu_timing`) for a 33-coefficient binary-polynomial multiplication. In total the 131-coefficient multiplication takes 14503 cycles (measured). This includes 11727 cycles for 9 33-coefficient multiplications, cycles required for combination of the results, and function-call overhead.

In total, a normal-basis multiplication including input conversion, 131-coefficient polynomial multiplication and output conversion with reduction takes 16635 cycles (measured).

Squaring. In normal basis a squaring is a cyclic shift of bits, so squaring consists of 131 loads and 131 stores to cyclically shifted locations. A call to the squaring function in normal-basis representation takes 328 cycles (measured).

***m*-Squaring.** The only difference between squaring and *m*-squaring is the shifting distance of the store locations. The implementation uses separate fully unrolled *m*-squaring functions for all relevant values of *m*. Each *m*-squaring therefore takes 328 cycles (measured), just like a squaring.

Computation of ϕ^j . The computation of ϕ^j cannot just simply be realized as a single *m*-squaring with $m = j$, because the value of *j* is most likely different for the 128 bitsliced values in one batch. Recall that the value of *j* is determined by three bits of the Hamming weight of the *x*-coordinate, denote these bits with b_1, b_2, b_3 such that $j = b_1 + 2b_2 + 4b_3 + 3$. Now the computation of $r \leftarrow \phi^j(x)$ can be carried out as follows:

```

r ← x23
if b1 then r ← r2
if b2 then r ← r22
if b3 then r ← r24.

```

When using bitsliced representation, conditional statements have to be replaced by equivalent arithmetic computations, a statement of the form

```
if b then r ← s
```

is performed as

$$r_i \leftarrow (r_i \& \neg b) \oplus (s_i \& b).$$

for each bitsliced bit r_i of *r* and s_i of *s*.

In total the computation of $\phi^j(x)$ requires one 3-squaring and three conditional *m*-squarings, each of which requires 393 logical instructions. One computation of ϕ^j takes 1380 cycles (measured).

Addition. Addition requires loading 262 inputs, 131 xors and storing of 131 outputs. Just as squaring, the function is bottlenecked by loads and stores rather than logical instructions. One call to the addition function takes 491 cycles (measured).

Inversion. Inversion is implemented using Fermat's little theorem. It involves 8 multiplications, 3 squarings and 6 m -squarings (with $m = 2, 4, 8, 16, 32, 65$). It takes 135460 cycles (measured). Observe that with a sufficiently large batch size for Montgomery inversion this does not have big impact on the cycle count of one iteration.

Hamming-weight computation. The bitsliced Hamming-weight computation of a 131-bit number represented in normal basis is done in a divide-and-conquer approach (producing bitsliced results) using 625 bit operations. The function is fully unrolled and uses 845 cycles (measured) to compute the Hamming weight.

Full iteration. In order to support a batch size of 512 for Montgomery inversions the batches are stored in main memory and fetched into the local storage temporarily for computations using DMA transfers. Without DMA transfers to and from main memory the maximal batch size would be 14 due to the limited size of the local storage.

Since the access pattern to the batches is deterministic, it is possible to use multi-buffering to prefetch data while processing previously loaded data and to write back data to main memory during ongoing computations. Even though 3 slots—one for outgoing data, one for computation, and one for incoming data—are sufficient for the buffering logic, the implementation uses 8 slots in local memory as ringbuffer to hide nondeterministic delays on the memory bus. One DMA tag is assigned to each of these slots to monitor ongoing transactions. At the beginning of the computation, one slot is chosen for the first batch and the batch is loaded into the local storage. During one step of the iteration function, the SPU iterates multiple times over the batches.

Each time the active set of data in the local storage changes, the SPU first determines the next slot in the ringbuffer and uses a blocking call to check whether previous writebacks from this slot to main memory have finished. It then initiates a prefetch for the next required batch into this slot. Then it uses another blocking call to wait for the current active set of data to be fully transferred from main memory to local storage. After this data is processed the SPU finally initiates a DMA transfer to write changed data back to main memory.

With the batch size of 512 for Montgomery inversions one iteration takes 75315 bit operations, the computation takes 94949 cycles on 128 parallel iterations (measured), which corresponds to 742 cycles per iteration. This cycle count does not include the cost for handling distinguished points when they are found. Writing the distinguished point and reading in a new input takes many cycles but occurs only very infrequently, so the total overhead is negligible. The overhead from loads, stores, function calls, and control flow is thus only about 27 percent.

An implementation without any DMA transfers to main memory and batch size for Montgomery inversions of 14 requires 80265 bit operations per iteration and takes 99993 cycles per iteration. This shows that indeed most overhead from DMA transactions can be hidden between computations. Measurements on IBM blade servers QS21 and QS22 showed that neither processor bus nor main memory are a bottleneck even if 8 SPEs are doing independent computations and DMA transfers in parallel.

6.4 Implementing the iteration function on NVIDIA GPUs

Today's graphics cards contain powerful graphics processing units (GPUs) to handle the increasing complexity and screen resolution in video games. GPUs are now powerful, highly parallel computing platforms with much computing power. GPUs have been used in different cryptographic and cryptanalytical contexts [CIKL05, CK06, MPS10, SG08, BCC⁺09].

In order to understand the implementation of the ECC2K-130 iteration function it is necessary to first understand the programming model and the various bottlenecks of graphics processing units; both are quite different from those of general-purpose CPUs. The following description will focus on the NVIDIA GTX 295, the target of the implementation. This graphics card contains two NVIDIA G200b GPUs.

6.4.1 Programming the NVIDIA GTX 295 graphics card

Modern GPUs are highly parallel computing platforms. NVIDIA's GTX 295 video card contains two G200b GPUs; each GPU contains 30 cores, each core contains 8 ALUs and each ALU is able to perform a 32-bit operation each cycle.

The most impressive feature of GPUs is their theoretical floating-point performance. Each of the 480 ALUs on a GTX 295 can perform a single-precision floating-point multiplication (with a free addition) every cycle. There are also 120 "special-function units"; each of these units can perform another single-precision floating-point multiplication per cycle. The GPU operates at 1.242 GHz so it can in total perform 745 billion floating-point multiplications per second. The most useful GPU arithmetic instructions for the ECC2K-130 computation are 32-bit logical instructions rather than floating-point multiplications, but 596 billion 32-bit logical instructions per second are still much more impressive than for example the 28.8 billion 128-bit logical instructions per second performed by a typical 2.4-GHz Intel Core 2 CPU with 4 cores and 3 128-bit ALUs per core.

This computational power of GPUs is exploited by running many hardware threads in parallel: Each core handles threads in groups of 32 called *warps* in a single-instruction-multiple-threads (SIMT) approach.

However, GPUs have many bottlenecks that make most applications run slower, often one or two orders of magnitude slower, than the theoretical throughput figures would suggest. The most troublesome bottlenecks are discussed in the remainder of this section.

The dispatcher. The 8 ALUs in a GPU core are fed by a single *dispatcher*. The dispatcher cannot issue more than one new instruction to the ALUs every 4 cycles. The dispatcher can send this one instruction to a *warp* containing 32 separate *threads* of computation, applying the instruction to 32 pieces of data in parallel and keeping all 8 ALUs busy for all 4 cycles; but the dispatcher cannot direct some of the 32 threads to follow one instruction while the remaining threads follow another.

Branching is allowed, but if threads within one warp take different branches, the threads taking one branch will no longer operate in parallel with the threads in the other branch; execution of the two branches is serialized and the time it takes to execute diverging branches is the sum of the time taken in all branches.

Instruction latency. Each thread follows its instructions strictly in order. NVIDIA does not document the exact pipeline structure but recommends to run at least 192 threads (6 warps) on each core to hide arithmetic latency. If 8 ALUs are fully occupied with 192 threads then each thread runs every 24 cycles; evidently the latency of an arithmetic instruction is below 24 cycles.

One might think that a single warp of 32 threads can keep the 8 ALUs fully occupied, if the instructions in each thread are scheduled for 24-cycle arithmetic latency (i.e., if an arithmetic result is not used until 6 instructions later). However, if only one warp is executed on one core, the dispatcher will issue instructions only every second dispatching cycle. Therefore at least 2 warps (64 threads) are necessary to exploit all ALU cycles.

Experiments showed that for sequences of arithmetic instructions the minimum number of threads that can keep the 8 ALUs fully occupied is 128. They furthermore showed a drop in the ALU utilization for 128 threads when more than 25% of the instructions are complex instructions that include memory access, or when complex instructions are adjacent.

NVIDIA also recommends running many more threads to hide memory latency. This does not mean that one can achieve the best performance by simply running the maximum number of threads that fit into the core. Threads share several critical resources, as discussed below, so increasing the number of threads means reducing the resources available to each thread. The ECC2K-130 computation puts extreme pressure on shared memory, as discussed later; to minimize this pressure it uses just 128 threads, which is just enough to avoid severe latency problems.

Registers and shared memory. Each core has 16384 32-bit registers, these registers are divided among the threads. For example, if the core is running 256 threads, then each thread is assigned 64 registers. If the core is running 128 threads, then each thread is assigned 128 registers, although the high 64 registers are somewhat limited: the architecture does not allow a high register as the second operand of an instruction, and does not allow read access to the last register, thus, only 127 registers are usable. Even with less than 128 threads, only 128 registers are available per thread.

Each core also has 16384 bytes of shared memory, which provides variable array indexing and communication between threads. This memory is split into 16 banks, each of which can dispatch one 32-bit read or write every two cycles. The 16 threads of one half-warp can execute a load operation in two cycles only if they all load from different memory banks or from exactly the same location. If n threads within one half-warp access different locations of the same memory bank in one load operation, they will typically take $2n$ cycles. Observe that this can lead to a factor-16 slowdown if all 16 threads of a half-warp load from the same memory bank.

Threads also have fast access to the 8192-byte constant memory. This memory can broadcast a 32-bit value from one location to every thread simultaneously, but it cannot read more than one location per cycle.

Global memory and local memory. The CPU makes data available to the GPU by copying it into the global memory on the graphics card. The cores on the GPU can then load data from this global memory and store results in global memory to be retrieved by the CPU. Global memory is also a convenient temporary storage area for data that does not fit into shared memory. However, global memory is limited to a throughput of just one 32-bit load from each core per cycle, with a latency of 400–600 cycles.

Each thread also has access to *local memory*. The name local memory might suggest that this storage is fast, but in fact it is as slow as global memory. Instructions accessing local memory automatically incorporate the thread ID into the address being accessed, effectively partitioning the local memory among threads without any extra address-calculation instructions.

There are no hardware caches for global memory and local memory. Programmers can, and must, set up their own schedules for copying data to and from global memory.

6.4.2 131-coefficient binary-polynomial multiplication

With optimal polynomial bases (see Subsection 6.2.3), each iteration involves five 131-coefficient binary-polynomial multiplications and only about 10000 extra bit operations. As described before the smallest known number of bit operations for multiplication of two 131-coefficient binary polynomials is 11961 [Ber09a, Section 2].

These figures show that polynomial multiplication consumes more than 80% of the bit operations in each iteration, putting a high priority on making multiplication run quickly; many decisions in the design of the implementation are driven by this primary target. This section describes these decisions and explains how the implementation carries out 131-coefficient binary-polynomial multiplication in less than 200 cycles on a single core.

The importance of avoiding DRAM. First experiments used an embarrassingly vectorized approach: T threads in a multiprocessor work on $32T$ independent multiplications in bitsliced form. The $32T \times 2$ inputs are stored as 262 vectors of $32T$ bits, and the $32T$ outputs are stored as 261 vectors of $32T$ bits.

The main difficulty with this approach is that, even if the outputs are perfectly overlapped with the inputs, even if no additional storage is required, the inputs cannot fit into registers and shared memory. For $T = 128$ the inputs consume 134144 bytes, while shared memory and registers together have only 81920 bytes. Reducing T to 64 (and tolerating 50% GPU utilization) would fit the inputs into 67072 bytes, but would also make half of the registers inaccessible (since each thread can access at most 128 registers), reducing the total capacity of shared memory and registers to 49152 bytes.

There is more than enough space in DRAM, even with very large T , but DRAM throughput then becomes a serious bottleneck. A single pass through the input vectors, followed by a single pass through the output vectors, keeps the DRAM occupied for $523T$ cycles (i.e., more than 16 cycles per multiplication), and any low-memory multiplication algorithm requires many such passes.

Different implementations of multiplication algorithms and complete iteration functions using this approach achieved a maximal speed of only 26 million iterations per second on a GTX 295. The remainder of this section describes a faster approach.

How to fit into shared memory. The SIMD programming model of GPUs highly relies on the exploitation of data-level parallelism. However, data-level parallelism does not require having each thread work on a completely independent computation: parallelism is also available within computations. For example, the addition of two 32-way-bitsliced field elements is nothing but a sequence of 131 32-bit xor operations; it naturally contains 131-way data-level parallelism. Similarly, there are many ways to break 131-coefficient binary-polynomial multiplication into several smaller polynomial multiplications that can be carried out in parallel.

Registers do not communicate between threads, so having several threads cooperate on a single computation requires the active data for the computation to fit into shared memory. On the other hand, registers have more space than shared memory; the multiplication in fact uses some registers as spill locations for data not involved in the multiplication, reversing the traditional direction of data spilling from registers to memory.

The final software carries out 128 independent 131-bit multiplications (i.e., four 32-way bitsliced 131-bit multiplications) inside shared memory and registers, with no DRAM access. This means that each multiplication has to fit within 1024 bits of shared memory. This would not have been a problem for schoolbook multiplication, but it was a rather tight fit for the fast Karatsuba-type multiplication algorithm described below; more simultaneous multiplications would have meant compromises in the multiplication algorithm.

The implementation uses 128 threads. This means that 32 threads are cooperating on each of the four 32-way bitsliced 131-bit multiplications. Experiments confirmed that these are enough threads to hide most latencies in the most time-consuming parts of the iteration function, particularly multiplication. The 131-bit multiplication algorithm allows close-to-32-way parallelization, as discussed below, although the parallelization is not perfect.

There would be fewer latency problems from 192 or 256 threads, but the overall benefit is overwhelmed by increased parallelization requirements within each multiplication. In the opposite direction, parallelization requirements could be reduced by running 96 or 64 threads, but below 128 threads the GPU performance drops almost linearly.

Vectorized 128-bit multiplication. The main task is now to multiply 131-bit polynomials, at each step using using 32 parallel bit operations to the maximum extent possible. The resulting algorithm is then repeated on 128 independent inputs to obtain what the code actually does with 128 threads: namely, 128 separate multipli-

cations of 131-bit polynomials, stored in bitsliced form as $4 \cdot 131$ 32-bit words, using 128 parallel 32-bit operations to the maximum extent possible.

The main part in the implementation consists in the slightly simpler task of multiplying 128-bit polynomials which is done by three levels of Karatsuba expansion. Each level uses $2n$ xors to expand a $2n$ -bit multiplication into three n -bit multiplications, and then $5n - 3$ xors to collect the results (with Bernstein's refined Karatsuba from [Ber09b]).

Three levels of Karatsuba result in 27 16-bit polynomial multiplications. The inputs to these multiplications occupy a total of 864 bits, consuming most but not all of the 1024 bits of shared memory available to each 131-bit multiplication. A self-written scheduler for the code from [Ber09b] for a 16-bit polynomial multiplication produces a sequence of 350 bit-logical instructions and 63 load/store instructions that uses 67 registers. This sequence is used to carry out the 27 multiplications in parallel leaving 5 threads idle out of 32. In total 108 16-bit polynomial multiplications coming from the four independent 131-bit polynomial multiplications are carried out by 108 threads.

The initial Karatsuba expansion is trivially parallelizable and is merged across all three levels, operating on blocks of 16 bits per operand and using 8 loads, 19 xors, and 27 stores per thread.

Karatsuba collection is more work: On the highest level (level 3), each block of 3 32-bit results (with leading coefficient zero) is combined into a 64-bit intermediate result for level 2. This takes 5 loads (2 of these conditional), 3 xors and 3 stores per thread on each of the 9 blocks. Level 2 operates on blocks of 3 64-bit intermediate results leading to 3 128-bit blocks of intermediate results for level 1. This needs 6 loads and 5 xors for each of the 3 blocks. The 3 blocks of intermediate results of this step do not need to be written to shared memory and remain in registers for the following final step on level 1. Level 1 combines the remaining three blocks of 128 bits to the final 256-bit result by 12 xors per thread.

Vectorized 131-bit multiplication. To multiply 131-bit polynomials each input is split into a 128-bit low part and a 3-bit high part, handling the 3×3 -bit and 3×128 -bit products separately.

The 3×3 -bit multiplication is carried out almost for free by an otherwise idle 16×16 multiplication thread. The 3×128 -bit multiplication is implemented straightforwardly by 6 loads for the 3 highest bits of each input, $3 \cdot 4$ combined load-and-ands per input, and 24 xors.

Overall the 131-coefficient polynomial multiplier uses 13087 bit operations, and about 40% of the ALU cycles are spent on these bit operations rather than on loads, stores, address calculations, and other overhead. An extra factor of about 1.1 is lost from 32-way parallelization, since the 32 threads are not always all active. For comparison, the Toom-type techniques from [Ber09b] use only 11961 bit operations, saving about 10%, but appear to be more difficult to parallelize.

6.4.3 ECC2K-130 iterations on the GPU

Recall that polynomial multiplication, the topic of the previous section, consumes more than 80% of the bit operations in the ECC2K-130 computation. This does not mean that the 20% overhead can be ignored. If, for example, the polynomial-multiplication code was carrying out useful bit operations in 40% of its cycles, while the remaining code was carrying out useful bit operations in only 10% of its cycles, the total time would be twice the polynomial-multiplication time.

This section discusses several aspects of the overhead in the ECC2K-130 computation and finally describes how the the different operations are combined to implement the ECC2K-130 iteration function. The main goal, as in the previous section, is to identify 32-way parallelism in the bit operations inside each 131-bit operation. This is often more challenging for the operations described in this section than it is for multiplication, and in some cases requires changes to the algorithms to improve parallelism. Inputs to all of the algorithms described in this section are in permuted-normal-basis representation.

Basis conversion. Before two elements in permuted-normal-basis representation can be multiplied they have to be converted to optimal-polynomial-basis representation.

Consider an element a of $\mathbb{F}_{2^{131}}$ in permuted normal basis:

$$a = a_0(\zeta + \zeta^{-1}) + a_1(\zeta^2 + \zeta^{-2}) + \cdots + a_{130}(\zeta^{131} + \zeta^{-131}).$$

Basis conversion is now carried out in 6 levels of computation. On levels 0 and 1 of the basis-conversion algorithm the following sequence of operations is executed on bits $a_0, a_{62}, a_{64}, a_{126}$:

$$\begin{aligned} a_{62} &\leftarrow a_{62} + a_{64} \\ a_0 &\leftarrow a_0 + a_{126} \\ a_{64} &\leftarrow a_{64} + a_{126} \\ a_0 &\leftarrow a_0 + a_{62} \end{aligned}$$

Meanwhile the same operations are performed on bits a_1, a_{61}, a_{65} , and a_{125} ; on bits a_2, a_{62}, a_{60} , and a_{124} ; and so on through bits a_{30}, a_{32}, a_{94} , and a_{96} . These 31 groups of bits are assigned to 32 threads, keeping almost all of the threads busy.

Merging levels 2 and 3 and levels 4 and 5 works in the same way. This assignment keeps 24 out of 32 threads busy on levels 2 and 3, and 16 out of 32 threads busy on levels 4 and 5. This assignment of operations to threads also avoids almost all memory-bank conflicts (see Subsection 6.4.1). A conversion to polynomial basis takes 1924 cycles.

Multiplication with reduction. Recall that a ppp multiplication produces a product in polynomial basis, suitable for input to a subsequent multiplication. A ppn multiplication produces a product in permuted normal basis, suitable for input to a squaring.

The main work in ppn, beyond polynomial multiplication, is a conversion of the product from polynomial basis to normal basis. This conversion is almost identical to the conversion to polynomial basis described above, except that it is double-size. The

main work in ppp is a slightly more complicated double-size conversion, with similar parallelization. One ppn multiplication takes 21083 cycles, a ppp multiplication takes 22362 cycles.

Squaring and m -squaring. Squaring and m -squaring in permuted-normal-basis representation are simply permutations.

The obvious method for 32 threads to permute 131 bits is for them to pick up the first 32 bits, store them in the correct locations, pick up the next 32 bits, store them in the correct locations, etc.; each thread performs 5 loads and 5 stores, with most of the threads idle for the final load and store. The addresses determined by the permutation for different m -squarings can be kept in constant memory. However, this approach triggers two GPU bottlenecks.

The first bottleneck is shared-memory bank throughput. Recall from Subsection 6.4.1 that threads in the same half-warp cannot simultaneously store values to the same memory bank. A self-written greedy search tool decides on a good order to pick up 131 bits, trying to avoid all memory bank conflicts for both the loads and the stores. For almost all values of m , including the most frequently used ones, this tool found a conflict-free assignment. For two values of m the assignment involves a few bank conflicts, but these values are used only in inversion.

The second bottleneck is constant-memory throughput. If each thread i loads from a constant array at position i then the constant cache serves only one thread per cycle. This bottleneck is eliminated by moving these loads out of the main loop and dedicating 10 registers per thread to hold 20 load and 20 store positions for the 4 most-often used m -squarings, packing 4 1-byte positions in one 32-bit register. Unpacking the positions costs just one shift and one mask instruction for the two middle bytes, a mask instruction for the low byte and a shift instruction for the high byte.

Computation of ϕ^j . The computation of ϕ^j is carried out in a similar way as in the Cell implementation, it also uses one 3-squaring and 3 conditional m -squarings. The conditional m -squarings with $m = 1, 2, 4$ use the table positions stored in registers and thus do not use any lookups from constant memory. Parallelization is the same as as for squarings and m -squarings described above. One computation of $\phi^j(x)$ takes 7283 cycles.

Addition. Adding 128 bitsliced elements of $\mathbb{F}_{2^{131}}$ is done by the 128 threads using 10 loads, 5 xors and 5 stores per thread. Out of these instructions, 2 loads, 1 xor and 1 store are conditional and only carried out by threads with thread ID smaller than 12.

Inversion. Inversion is implemented using the same sequence of multiplications, squarings, and m -squarings as on the Cell processor. It uses 3 squarings, 6 m -squarings (with $m = 2, 4, 8, 16, 32, 65$), 15 conversions to polynomial basis, 7 ppn multiplications, and 1 ppp multiplication. One inversion takes 225116 cycles.

Hamming-weight computation. As in the Cell implementation the Hamming-weight computation receives a bitsliced input and computes a bitsliced output. More specifically, the first 8 bits of the input value x are overwritten with bits h_0, \dots, h_7 such that

the Hamming weight of the input value x is $\sum_{i=0}^7 h_i 2^i$. During the parallel computation of these 8 bits also other bits of x are overwritten. The basic building block for the parallel computation is a full adder, which has three input bits b_1, b_2, b_3 and uses 5 bit operations to compute 2 output bits c_0, c_1 such that $b_1 + b_2 + b_3 = c_1 2 + c_0$. When the full adder overwrites one of the input bits with c_1 this bit gets a weight of 2. If three such bits with a weight of 2 are input to a full adder, one of the output bits will have a weight of 4. More generally: If three bits with a weight of 2^i enter a full adder, one output bit will have a weight of 2^i , the other one a weight of 2^{i+1} . At the beginning of the computation all 131 bits have a weight of 2^0 .

Because there are many input bits, it is easy to keep many threads active in parallel. In the first addition round 32 threads perform 32 independent full-adder operations, 96 bits with weight 2^0 are transformed into 32 bits with weight 2^0 and 32 bits with weight 2^1 . This leaves $131 - 96 + 32 = 67$ bits of weight 2^0 and 32 bits of weight 2^1 .

In the second round, 22 threads pick up 66 bits of weight 2^0 and produce 22 bits of weight 2^0 and 22 bits of weight 2^1 . At the same time 10 other threads pick up 30 bits of weight 2^1 and produce 10 bits of weight 2^1 and 10 bits of weight 2^2 . This leaves $67 - 66 + 22 = 23$ bits of weight 2^0 , $32 - 30 + 22 + 10 = 34$ bits of weight 2^1 , and 10 bits of weight 2^2 .

In the third round 7 threads perform full-adder operations on 21 input bits with weight 2^0 , 11 threads perform full-adder operations on 33 input bits of weight 2^1 , and 3 threads perform full-adder operations on 9 input bits of weight 2^2 .

This parallel computation needs 13 rounds to compute the bits h_0, \dots, h_7 , i.e. 8 bits with weight $2^0, \dots, 2^7$. The implementation actually uses a somewhat less parallel approach with 21 rounds, two of these rounds being half-adder operations which receive only 2 input bits and take only 2 bit operations. This has the benefit of simplifying computation of the input positions as a function of the thread ID.

Full iteration. Calling (launching) a GPU kernel from the CPU takes several microseconds on top of any time needed to copy data between global memory and the CPU. These costs are eliminated by running a single kernel for several seconds. The kernel consists of a loop around a complete iteration; it performs the iteration repeatedly without contacting the CPU. Any distinguished points are masked out of subsequent updates; distinguished points are rare, so negligible time is lost computing unused updates.

The kernel streams a batch of iterations in a simple way between global memory and shared memory; this involves a small number of global-memory copies in each iteration, these account for about 10% of the cycles per iteration. The kernel does not spill any additional data to DRAM; in particular, it avoids all use of local memory.

All of this sounds straightforward but in fact required completely redesigning the programming environment: Programs for NVIDIA GPUs are usually written in the C-like CUDA language, and translated by NVIDIA's `nvcc` compiler into a `*.ptx` file in a somewhat machine-independent PTX language. This `*.ptx` file is further translated by NVIDIA's `ptxas` compiler into a machine-specific binary `*.cubin` file. The `*.cubin` file is loaded onto the GPU and run. A notable difference between

programs running on GPUs and programs running on CPUs is that all functions implemented in `*.cu` files get inlined by `nvcc` to build one fully-inlined kernel.

NVIDIA's register allocators were designed to handle small kernels consisting of hundreds of instructions; their memory-use scaling appears to be quadratic with the kernel size, and their time scaling appears to be even worse. For medium-size kernels NVIDIA's compilers are intolerably slow; even worse, the resulting code involved frequent spills to local memory, dropping performance by an order of magnitude. For larger kernels the compilers ran out of memory (on a machine with 8 GB of RAM) and crashed.

Writing code in NVIDIA's PTX language did not solve this problem because PTX still requires compilation by `ptxas`; even though `ptxas` is labeled as an assembler it turns out to be the culprit in NVIDIA's register-allocation problems. To control register allocation the whole kernel is implemented in `qasm` (see Section 2.8) and then compiled to the reverse-engineered assembler `cuasm` by van der Laan [vdL07]. According to [WPSAM10] the level-3 instruction cache of the G200b GPU has a size of 32 KB and is shared between 3 cores. A fully unrolled version of the code consists of more than 125,000 instructions (about 1 MB) when using a batch size of 32 for Montgomery inversions. Using non-unrolled loops allows to increase the batch size to 128. The resulting code still does not fit into the level-3 instruction cache, incurring significant performance penalty.

The GPU instruction set contains a `call` instruction which can be used to implement function calls. Using such function calls to frequently used code blocks—in particular 131-coefficient polynomial multiplication—the code size is further reduced to 5202 instructions, almost fitting into the level-3 instruction cache.

With a batch size of 128 for Montgomery inversions on top of 128 parallel bit-sliced streams of computation one iteration takes 1164 cycles.

6.5 Performance results and comparison

Both the implementation for the SPUs of the Cell Broadband Engine and the implementation for GPUs use a bitsliced approach and perform all arithmetic operations on 128 inputs in parallel so a lower bound of the cycles per iteration can be derived from the number of bit operations per iteration. Although each multiprocessor of the G200b GPU can theoretically perform 256 bit operations (8 logical instructions on 32-bit words) and each SPU can perform only 128 bit operations per cycle, the number of cycles are higher on the GPU than on the Cell for all operations summarized in Table 6.1. There are several reasons that the code on the SPU gets so much closer to the lower bound defined by bit operations than the GPU:

- On the GPU load and store instructions are not free as on the Cell where in most cases they can be interleaved with logical instructions;
- carrying out computations on non-independent data by multiple threads costs additional instructions to compute indices based on the thread ID;
- not all 128 threads can be kept active all the time;

- 128 threads are enough to avoid severe latency problems, but are not enough to hide all latencies;
- threads are synchronized only within one warp, synchronizing threads across warps to avoid data hazards when accessing shared memory requires additional cycles;
- an overhead of about 10% is incurred by data transfers between global and shared memory;
- the size of the kernel is still slightly larger than the level-3 instruction cache; this leads to instruction-cache misses that degrade performance further.

Despite all these overheads a single NVIDIA GTX 295 graphics card is able to perform more than twice as many iterations per second as a Sony PlayStation 3: The 60 multiprocessors of the two G200b GPUs running at 1242 MHz perform 64.00 million iterations per second; the 6 usable SPUs of the Cell Broadband Engine in a PlayStation 3 running at 3192 MHz perform 25.82 million iterations per second.

Detailed performance numbers of the building blocks of both implementations are given in Table 6.1. The performance figures for the Cell Broadband Engine were measured on a PlayStation 3 named `stan`; details of this machine are given in Table 4.1. The measurements on the Cell were obtained by computing the average of 50 function calls. These averages are very stable as only one process can run on the SPU, access to the local storage is uncached and the SPUs use static branch prediction. Each function call performs 128 parallel bitsliced operations; Table 6.1 reports the cycles taken by one operation, i.e. the above average divided by 128. The overall performance of the iteration function ignores cost for outputting distinguished points and reading in new input points.

Benchmarks on the GPU cannot be obtained in the same way because the routines are all inlined and not implemented as separate functions. These numbers are instead obtained as follows: On a typical pass through the main loop (specifically, the 10000th pass), inside each thread, read the GTX 295's hardware cycle half-counter before and after each use of the routine and tally the cycles spent inside the routine. Repeat this experiment 20 times, with 128 threads in each experiment, yielding a total of $20 \cdot 128 = 2560$ cycle counts. To obtain numbers comparable to the measurements on the Cell Broadband Engine, each of these counts is divided by the number of "calls" to the respective routine and then divided by 128. Unlike for the Cell Broadband Engine the 2560 cycle counts contain a significant variation. This is due to the fact that threads running on one core do not necessarily take the same time inside the same operation. Table 6.1 reports average cycles and standard deviation from this average for the building blocks of the GPU implementation.

The expected total number of iterations required to solve the ECDLP given in the ECC2K-130 challenge is $2^{60.9}$ [BBB⁺09b]; together with the number of cycles per iteration on Cell SPU and NVIDIA GTX 295 respectively, this yields an estimate of the time it would take to solve the ECC2K-130 challenge: Running the software described in Section 6.3 on all 6 SPUs of Sony PlayStation 3 gaming consoles in parallel, the

Table 6.1: Cycle counts and bit-operations for the building blocks used by the Cell and the GPU implementations of the ECC2K-130 iteration function

operation	bit operations	G200b core	CBE-SPU
normal-basis multiplication	14249	—	129.96
opt.-poly.-basis multiplications			—
ppn multiplication	13498	159.54 ± 1.64	—
ppp multiplication	13506	158.08 ± 0.23	—
conversion to poly. basis	325	12.63 ± 0.05	—
squaring / m -squaring	0	9.60 ± 0.12	2.56
compute $\phi^j(x)$	1179	44.99 ± 0.83	10.78
Hamming-weight computation	625	41.60 ± 0.11	6.60
test $\text{HW}(x) < 34$	29	11.26 ± 0.16	6.59
addition	131	4.01 ± 0.06	3.84
conditional addition	262	6.81 ± 0.77	—
inversion			
normal basis	113992	—	1058.28
opt. poly. basis	112867	1758.72 ± 20.48	—
Full iteration			
$B = 128$, opt. poly. basis	73825	1164.43 ± 12.05	—
$B = 14$, normal basis	80264	—	781.20^a
$B = 512$, normal basis	75315	—	741.79^b

^a Compiled with `spu-gcc -O3 -march=cell -mtune=cell -Wall -fschedule-insns -mstdmain -fomit-frame-pointer`

^b Compiled with `spu-gcc -O3 -march=cell -mtune=cell -Wall -fschedule-insns -mstdmain`

challenge could be solved in an expected time of one year with 2643 PlayStations. Running the software described in Section 6.4 on all 60 cores of NVIDIA GTX 295 video cards in parallel, the challenge could be solved in an expected time of one year with 1066 video cards. As a comparison, the challenge could also be solved in an expected time of one year using 3190 quad-core AMD Phenom 955 X4 processors with the implementation described in [BBB⁺09b], or using 616 XC3S5000-4FG676 FPGAs with the implementation described in [FBB⁺10].

7

Implementing Wagner’s generalized birthday attack

Wagner’s generalized birthday algorithm [Wag02a, Wag02b] is a cryptanalytical algorithm that typically has performance bottlenecks very different from those addressed in the previous chapters. Instead of CPU performance it is hard-disk and network throughput that determines the performance of this algorithm for cryptographically relevant input sizes.

The implementation described in this chapter mounts the attack against the compression function of the toy version FSB_{48} of the Fast Syndrome-Based hash function (FSB) introduced by Augot, Finiasz and Sendrier in [AFS03] and submitted to the NIST SHA-3 competition [AFG⁺09]. The attack is carried out on a computer cluster with 5.5 TB of distributed storage, which is not enough for a straightforward realization of the attack. This lack of storage reflects the situation for most cryptanalytical applications of Wagner’s generalized birthday attack.

Section 7.1 explains Wagner’s generalized birthday attack and describes techniques to mount this attack in storage-restricted environments. Section 7.2 gives the necessary background of the target of the attack, namely the compression function of FSB and explains how it can be attacked using Wagner’s algorithm. The main part of this chapter consists of the description of the attack strategy in Section 7.3 and implementation details in Section 7.4. Finally Section 7.5 gives timing results and estimates of how long a similar attack would take against the full versions of FSB.

This chapter is based on joint work with Bernstein, Lange, Niederhagen, and Peters published in [BLN⁺09]. Aside from some minor changes to notation and phrasing the main differences between the content of this chapter and [BLN⁺09] are the following:

- a brief discussion of the efficiency of information-set decoding as an attack against the compression function of FSB is contained in [BLN⁺09] but omitted in this chapter; and
- this chapter contains a remark on the time it takes to find a collision in the full hash function FSB₄₈ with Floyd's cycle-finding algorithm.

7.1 Wagner's generalized birthday attack

The classic birthday problem considers the probability that two people out of a group of n people (for example a school class) have the same birthday. A slightly different question could be described as the “boys-girls birthday problem”: What is the probability that in a group of n girls and m boys at least one boy and one girl have the same birthday? For $n = m$ this boys-girls birthday problem corresponds to the easiest instance of the generalized birthday problem which is the following:

Given 2^{i-1} lists of B -bit strings, find 2^{i-1} strings, exactly one per list, such that their xor is zero.

For just two lists L_0 and L_1 this problem can be solved by sorting L_0 and then checking for each element in L_1 whether it is also in L_0 . For the case $i > 2$ Wagner gave an algorithm in [Wag02a]. This algorithm became known as Wagner's generalized birthday attack, Wagner's generalized birthday algorithm or Wagner's tree algorithm.

7.1.1 Wagner's tree algorithm

Wagner's algorithm builds a binary tree starting from the input lists. Let $L_{j,k}$ denote list k on level j of this tree; the input lists are $L_{0,0}, L_{0,1}, \dots, L_{0,2^{i-1}-1}$. For analysis of complexity and success probability of the algorithm assume that each list contains $2^{B/i}$ elements chosen uniformly at random from $[0, \dots, 2^B - 1]$.

The algorithm works as follows: On level 0 take the first two lists $L_{0,0}$ and $L_{0,1}$ and compare their list elements on the least significant B/i bits. Given that each list contains about $2^{B/i}$ elements, $2^{B/i}$ pairs of elements are expected to be equal on those least significant B/i bits. Take the xor of these pairs on all their B bits and put it into a new list $L_{1,0}$. Similarly compare the other lists—always two at a time—and look for elements matching on their least significant B/i bits which are xored and put into new lists. This process of *merging* yields 2^{i-2} lists containing each about $2^{B/i}$ elements which are zero on their least significant B/i bits. This completes level 0.

On level 1 take the first two lists $L_{1,0}$ and $L_{1,1}$ which are the results of merging the lists $L_{0,0}$ and $L_{0,1}$ as well as $L_{0,2}$ and $L_{0,3}$ from level 0. Compare the elements of $L_{1,0}$ and $L_{1,1}$ on their least significant $2B/i$ bits. As a result of the xoring in the previous level, the last B/i bits are already known to be 0, so it suffices to compare the next B/i bits. Since each list on level 1 contains about $2^{B/i}$ elements the expected number of elements that match on these next B/i bits is again $2^{B/i}$. The xor of each pair of

matching elements is put it into a new list $L_{2,0}$. Similarly compare the remaining lists on level 1.

Continue in the same way until level $i - 2$. On each level j consider the elements on their least significant $(j + 1)B/i$ bits of which jB/i bits are known to be zero as a result of the previous merge. Level $i - 2$ produces two lists containing about $2^{B/i}$ elements. The least significant $(i - 2)B/i$ bits of each element in both lists are zero. Comparing the elements of both lists on their $2B/i$ remaining bits gives 1 expected match, i.e., one xor equal to zero. Since each element is the xor of elements from the previous steps this final xor is the xor of 2^{i-1} elements from the original lists and thus a solution to the generalized birthday problem.

7.1.2 Wagner in storage-restricted environments

A 2007 paper [Ber07] by Bernstein includes two techniques to mount Wagner's attack on computers that do not have enough storage to hold all list entries. Various special cases of the same techniques also appear in a 2005 paper by Augot, Finiasz, and Sendrier [AFS05] and in a 2009 paper by Minder and Sinclair [MS09].

Clamping through precomputation. Suppose that there is space for lists of size only 2^b with $b < B/i$. Bernstein suggests to generate $2^{b+(B-ib)}$ entries per list and only consider those of which the least significant $B - ib$ bits are zero.

This idea can be generalized as follows: The least significant $B - ib$ bits can have an arbitrary value. This *clamping value* does not even have to be the same on all lists as long as the *sum* of all clamping values is zero. This is important if an attack does not produce a collision; in this case simply restart the attack with different clamping values.

Clamping through precomputation may be limited by the maximal number of entries that can be generated per list. Furthermore, halving the available storage space increases the precomputation time by a factor of 2^i .

Note that clamping some bits through precomputation might be a good idea even if enough storage is available as it can reduce the amount of data in later steps and thus make those steps more efficient.

After the precomputation step Wagner's tree algorithm is applied to lists containing bit strings of length B' where B' equals B minus the number of clamped bits. Performance evaluation considers lists on level 0 only *after* clamping through precomputation and then uses B instead of B' for the number of bits in these entries.

Repeating the attack. Another way to mount Wagner's attack in storage-restricted environments is to carry out the whole computation with smaller lists leaving some bits at the end "uncontrolled"; repeatedly running the attack with different clamping values then deals with the lower success probability.

In the context of clamping through precomputation each repeated run can simply vary the clamping values used during precomputation. If for some reason no bits can be clamped through precomputation the same idea of changing clamping constants can be applied in an arbitrary merge step of the tree algorithm. Note that any solution to the generalized birthday problem can be found by some choice of clamping values.

Expected number of runs. Wagner's algorithm, without clamping through precomputation, produces an expected number of exactly one collision. However this does not mean that running the algorithm necessarily produces a collision.

In general, the expected number of runs of Wagner's attack is a function of the number of remaining bits in the entries of the two input lists of the last merge step and the number of elements in these lists.

Assume that b bits are clamped on each level and that lists have length 2^b . Then the probability to have at least one collision after running the attack once is

$$P_{\text{success}} = 1 - \left(\frac{2^{B-(i-2)b} - 1}{2^{B-(i-2)b}} \right)^{2^{2b}},$$

and the expected number of runs $E(R)$ is

$$E(R) = \frac{1}{P_{\text{success}}}. \quad (7.1)$$

For larger values of $B - ib$ the expected number of runs is about 2^{B-ib} . The total time for the attack t_W is assumed to be linear in the amount of data on level 0, i.e.,

$$t_W \in \Theta \left(2^{i-1} 2^{B-ib} 2^b \right). \quad (7.2)$$

Here 2^{i-1} is the number of lists, 2^{B-ib} is approximately the number of runs, and 2^b is the number of entries per list. Observe that this formula usually underestimates the real time of the attack by assuming that all computations on subsequent levels are together still linear in the time required for computations on level 0.

Using Pollard iteration. If because of storage restrictions the number of uncontrolled bits is high, it may be more efficient to use a variant of Wagner's attack that uses Pollard iteration [Pol78], [Knu98, Chapter 3, exercises 6 and 7].

Assume that $L_0 = L_1$, $L_2 = L_3$, etc., and that combinations $x_0 + x_1$ with $x_0 = x_1$ are excluded. The output of the generalized birthday attack is then a collision between two distinct elements of $L_0 + L_2 + \dots$.

Another approach is to start with only 2^{i-2} lists L_0, L_2, \dots and apply the usual Wagner tree algorithm, with a nonzero clamping constant to enforce the condition that $x_0 \neq x_1$. The number of clamped bits before the last merge step is now $(i - 3)b$. The last merge step produces 2^{2b} possible values, the smallest of which has an expected number of $2b$ leading zeros, leaving $B - (i - 1)b$ uncontrolled.

Think of this computation as a function mapping clamping constants to the final $B - (i - 1)b$ uncontrolled bits and apply Pollard iteration to find a collision between the output of two such computations; combination then yields a collision of 2^{i-1} vectors.

As Pollard iteration has square-root running time, the expected number of runs for this variant is $2^{B/2-(i-1)b/2}$, each taking time $2^{i-2}2^b$ (compare (7.2)), so the expected running time is

$$t_{PW} \in \Theta \left(2^{i-2} 2^{B/2-(i-1)b/2+b} \right). \quad (7.3)$$

The Pollard variant of the attack becomes more efficient than plain Wagner with repeated runs if $B > (i + 2)b$.

7.2 The Fast Syndrome-Based hash function (FSB)

This section briefly describes the construction of the FSB hash function, omitting details that are necessary for implementing the function but do not influence the attack. The second part of this section gives a first idea of how Wagner’s generalized birthday attack can be used to find collisions of the compression function of FSB.

7.2.1 Details of the FSB hash function

The Fast Syndrome-Based hash function (FSB) was introduced by Augot, Finiasz and Sendrier in 2003. See [AFS03], [AFS05], and [AFG⁺09]. The security of FSB’s compression function relies on the difficulty of the “Syndrome Decoding Problem” from coding theory.

The FSB hash function processes a message in three steps: First the message is converted by a so-called domain extender into suitable inputs for the compression function which digests the inputs in the second step. In the third and final step the Whirlpool hash function designed by Barreto and Rijmen [BR01] is applied to the output of the compression function in order to produce the desired length of output. Wagner’s generalized birthday attack targets the compression function.

The compression function. The main parameters of the compression function are called n , r and w : consider n strings of length r which are chosen uniformly at random and can be written as an $r \times n$ binary matrix H . Note that the matrix H can be seen as the parity check matrix of a binary linear code. The FSB proposal [AFG⁺09] actually specifies a particular structure of H for efficiency; this structure is not exploited by the attack.

An n -bit string of weight w is called *regular* if there is exactly a single 1 in each interval $[(i - 1)\frac{n}{w}, i\frac{n}{w} - 1]_{1 \leq i \leq w}$. Such an interval is called a *block*. The input to the compression function is a regular n -bit string of weight w .

The compression function works as follows: The matrix H is split into w blocks of n/w columns. Each non-zero entry of the input bit string indicates exactly one column in each block. The output of the compression function is an r -bit string which is produced by computing the xor of all the w columns of the matrix H indicated by the input string.

Preimages and collisions. Finding a *preimage* to an output x of one round of the compression function means finding a regular n -bit string of weight w such that the compression function applied to this string yields x . A *collision* occurs if there are $2w$ columns of H —exactly two in each block—which add up to zero.

Finding preimages or collisions means solving two problems coming from coding theory: finding a preimage means solving the Regular Syndrome Decoding problem

and finding collisions means solving the so-called 2-regular Null-Syndrome Decoding problem. Both problems were defined and proven to be NP-complete in [AFS05].

Parameters. Following the notation in [AFG⁺09] the version of FSB which produces a hash value of length `length` is denoted $\text{FSB}_{\text{length}}$. Note that the output of the compression function has r bits where r is considerably larger than `length`.

For SHA-3 submissions, NIST required hash lengths of 224, 256, 384, and 512 bits, respectively. The SHA-3 proposal contains five versions of FSB: FSB_{160} , FSB_{224} , FSB_{256} , FSB_{384} , and FSB_{512} . The parameters for those versions are listed in Table 7.1.

The proposal also contains FSB_{48} , which is a reduced-size version of FSB and the main attack target here. The binary matrix H for FSB_{48} has dimension $192 \times 3 \cdot 2^{17}$; i.e., r equals 192 and n is $3 \cdot 2^{17}$. In each round a message chunk is converted into a regular $3 \cdot 2^{17}$ -bit string of Hamming weight $w = 24$. The matrix H contains 24 blocks of length 2^{14} . Each 1 in the regular bit string indicates exactly one column in a block of the matrix H . The output of the compression function is the 192-bit xor of those 24 columns.

A pseudo-random matrix. The attack against FSB_{48} uses a pseudo-random matrix H which is constructed as described in [AFG⁺09, Section 1.2.2]: H consists of 2048 submatrices, each of dimension 192×192 . For the first submatrix consider a slightly larger matrix of dimension 197×192 . Its first column consists of the first 197 digits of π where each digit is taken modulo 2. The remaining 191 columns of this submatrix are cyclic shifts of the first column. The matrix is then truncated to its first 192 rows which form the first submatrix of H . For the second submatrix consider digits 198 up to 394 of π . Again build a 197×192 bit matrix where the first column corresponds to the selected digits (each taken modulo 2) and the remaining columns are cyclic shifts of the first column. Truncating to the first 192 rows yields the second block matrix of H . The remaining submatrices are constructed in the same way.

This is one possible choice for the matrix H . The attack described in the following does not make use of the structure of this particular matrix. The reason to use this particular matrix in the implementation is that it is also contained in the FSB reference implementation submitted to NIST by the FSB designers.

7.2.2 Attacking the compression function of FSB_{48}

Coron and Joux pointed out in [CJ04] that Wagner's generalized birthday attack can be used to find preimages and collisions in the compression function of FSB. The following paragraphs present a slightly streamlined version of the attack of [CJ04] in the case of FSB_{48} .

Determining the number of lists for a Wagner attack on FSB_{48} . A collision for FSB_{48} is given by 48 columns of the matrix H which add up to zero; the collision has exactly two columns per block. Each block contains 2^{14} columns and each column is a 192-bit string.

The attack uses 16 input lists to solve this particular 48-sum problem. Each list entry is the xor of three columns coming from one and a half blocks. This ensures

that there are no overlaps, i.e., more than two columns coming from one matrix block in the end. Applying Wagner's attack in a straightforward way means that each list has at least $2^{\lceil 192/5 \rceil}$ entries. Clamping away 39 bits in each step yields an expected collision after one run of the tree algorithm.

Building lists. The input to Wagner's algorithm are 16 lists containing 192-bit strings, each being the xor of three distinct columns of the matrix H . Each triple of three columns is selected from one and a half blocks of H in the following way:

List $L_{0,0}$ contains the sums of columns i_0, j_0, k_0 , where columns i_0 and j_0 come from the first block of 2^{14} columns, and column k_0 is picked from the following block with the restriction that it is taken from the first half of it. There are about 2^{27} sums of columns i_0 and j_0 coming from the first block. These two columns are then added to all possible columns k_0 coming from the first 2^{13} elements of the second block of the matrix H , yielding a total of about 2^{40} elements for $L_{0,0}$.

Observe that splitting every second block in half neglects several solutions of the 48-xor problem. For example, a solution involving two columns from the first half of the second block cannot be found by this algorithm. However, fewer lists would require more storage and a longer precomputation phase to build the lists.

The second list $L_{0,1}$ contains sums of columns i_1, j_1, k_1 , where column i_1 is picked from the second half of the second block of H and j_1 and k_1 come from the third block of 2^{14} columns. This again yields about 2^{40} elements. The lists $L_{0,2}, L_{0,3}, \dots, L_{0,15}$ are constructed analogously.

For each list this procedure generates more than twice the amount of list entries needed for a straightforward attack as explained above; about $2^{40}/4$ of these entries are expected to be zero on their least significant two bits. Clamping those two bits away should thus yield a list of 2^{38} strings of length 190. Now a straightforward application of Wagner's attack applied to 16 lists with about $2^{190/5}$ elements is expected to yield a collision after completing the tree algorithm.

7.3 Attack strategy

This section analyzes how much storage is required to carry out Wagner's attack against the compression function of FSB_{48} , and describes the techniques used to mount the attack on a cluster with only 5.5 TB of storage.

7.3.1 How large is a list entry?

The number of bytes required to store one list entry depends on the representation of the entry.

Value-only representation. The obvious way of representing a list entry is as a 192-bit string, the xor of columns of the matrix. Bits already known to be zero of course do not have to be stored, so on each level of the tree the number of bits per entry decreases by the number of bits clamped on the previous level. Ultimately, the output of the algorithm has to be not be the *value* of the entry—after a successful attack it will

be all-zero—but the column positions in the matrix that lead to this all-zero value. However, Subsection 7.3.3 will show that computations only involving the *value* can be useful if the attack has to be run multiple times due to storage restrictions.

Value-and-positions representation. If enough storage is available just store positions in the matrix alongside the value. Observe that unlike storage requirements for *values* the number of bytes for *positions* increases in each entry with increasing levels, and becomes dominant for higher levels.

Compressed positions. Instead of storing full positions, storing positions modulo, for example, 256 saves storage. After the attack has successfully finished the full position information can be computed by checking which of the possible positions lead to the appropriate intermediate results on each level.

Dynamic recomputation. When keeping full positions the value does not have to be stored at all. Every time the value (or parts of it) is required it can be dynamically recomputed from the positions. In each level the size of a single entry doubles (because the number of positions doubles), the expected number of entries per list remains the same but the number of lists halves, so the total amount of data is the same on each level when using dynamic recomputation. As discussed in Section 7.2 there are 2^{40} possibilities to choose columns to produce entries of a list, so the positions on level 0 can be encoded in 40 bits (5 bytes).

Observe that it is possible to switch between certain representations during computation if at some level another representation becomes more efficient: from value-and-position representation to compressed-positions representation and back, from one of the above to compressed positions, and from any representation to value-only representation.

7.3.2 What list size can be handled with 5.5 TB of storage?

To estimate the storage requirements it is convenient to consider *dynamic recomputation* (storing positions only) because in this case the amount of required storage is constant over all levels and this representation has the smallest memory consumption on level 0.

As described in Subsection 7.2.2 the attack can start with 16 lists of size 2^{38} , each containing bit strings of length $r' = 190$. However, storing 16 lists with 2^{38} entries, each entry encoded in 5 bytes requires 20 TB of storage space.

The computer cluster used for the attack consists of eight nodes with a storage space of 700 GB each. Hence, the parameters of the attack have to change to cope with total storage limited to 5.5 TB.

On the first level there are 16 lists and each list entry needs at least 5 bytes. This means that at most $5.5 \cdot 2^{40} / 16 / 5 = 1.1 \times 2^{36}$ entries per list fit into 5.5 TB of storage. Some of the disk space is used for the operating system and so a straightforward implementation would use lists of size 2^{36} .

Clamping 4 bits in the 2^{40} generated entries during precomputation yields lists of expected size 2^{36} . Entries in these lists have a length of 188 bits (represented

through 5 bytes holding the positions from the matrix). Clamping 36 bits in each of the 3 steps leaves two lists of length 2^{36} with 80 non-zero bits. According to (7.1) this attack approach requires 256.5 expected repetitions to find a collision. The only way of increasing the list size to 2^{37} and thus reducing the number of runs is to use value-only representation on higher levels.

7.3.3 The strategy

The main idea of the attack strategy is to distinguish between the task of finding clamping constants that yield a final collision and the task of actually computing the collision.

Finding appropriate clamping constants. The task of just finding clamping constants that yield a collision does not require storing the matrix positions. Whenever storing the value needs less space than storing the matrix positions, the list entries can thus be compressed to value-only representation. As a side effect this speeds up the computations because less data has to be loaded and stored. Based on this observation the first phase of the attack is carried out as follows:

Starting from lists $L_{0,0}, \dots, L_{0,7}$, each containing 2^{37} entries first compute list $L_{3,0}$ (see Figure 7.1) on the eight cluster nodes. This list has entries with 78 remaining bits each. Section 7.4 will explain how these entries are presorted on hard disk according to 9 bits that do not have to be stored. Another 3 bits are determined by the node holding the data (see also Section 7.4) so only 66 bits or 9 bytes of each entry have to be stored, yielding a total storage requirement of 1152 GB versus 5120 GB necessary for storing entries in positions-only representation.

Continue with the computation of list $L_{2,2}$, which has entries of 115 remaining bits. Again 9 of these bits do not have to be stored due to presorting, 3 are determined by the node, so only 103 bits or 13 bytes have to be stored, yielding a storage requirement of 1664 GB instead of 2560 GB for uncompressed entries.

After these lists have been stored persistently on disk, proceed with the computation of list $L_{2,3}$, then $L_{3,1}$ and finally check whether $L_{4,0}$ contains at least one element. These computations require another 2560 GB.

The total amount of storage sums up to $1152 \text{ GB} + 1664 \text{ GB} + 2560 \text{ GB} = 5376 \text{ GB}$; obviously all data fits onto the hard disk of the eight nodes.

If a computation with given clamping constants is not successful, change clamping constants only for the computation of $L_{2,3}$. The lists $L_{3,0}$ and $L_{2,2}$ do not have to be computed again. All combinations of clamping values for lists $L_{0,12}$ to $L_{0,15}$ summing up to 0 are allowed. Therefore there are a large number of valid clamp-bit combinations.

With 37 bits clamped on every level and 3 clamped through precomputation only 4 bits are left uncontrolled and therefore, according to (7.1), the expected number of repetitions to find a collision is only 16.5.

Computing the matrix positions of the collision. After an appropriate set of clamping constants has been found in phase 1 of the attack and the values in $L_{3,0}$ and $L_{3,1}$

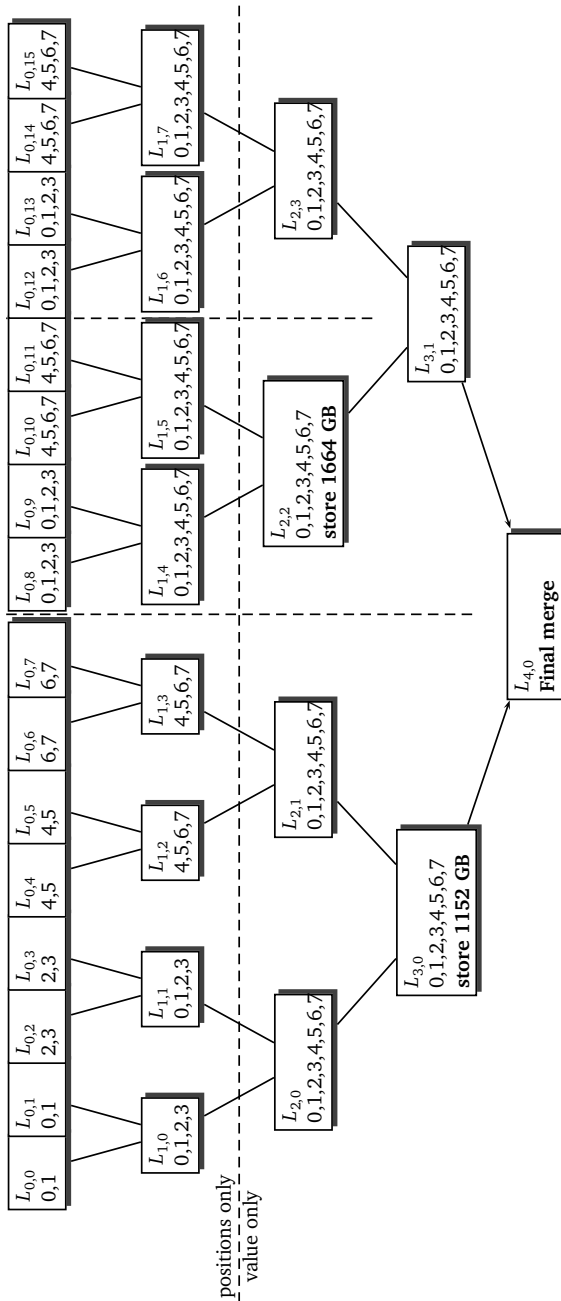


Figure 7.1: Structure of the generalized birthday attack against FSB_{48} : in each box the upper line denotes the list, the lower line gives the nodes holding fractions of this list

yielding a collision are known, phase 2 recomputes lists $L_{3,0}$ and $L_{3,1}$ and keeps track of matrix positions.

For this task all list entries are encoded in positions-only representation. On level 0 and level 1 this is the most space-efficient approach. Switching to compressed-positions representation on higher levels would not result in a significant speedup. In total each of the two half-tree computations requires 5120 GB of storage, so they have to be performed one after the other on eight nodes.

7.4 Implementing the Attack

The computation platform for this particular implementation of Wagner's generalized birthday attack on FSB are eight nodes of the Coding and Cryptography Computer Cluster at Eindhoven University of Technology [Lan]. Each node has an Intel Core 2 Quad Q6600 CPU with a clock rate of 2.40 GHz and direct fully cached access to 8 GB of RAM. About 700 GB mass storage are provided by a Western Digital SATA hard disk with 20 GB reserved for system and user data. The nodes are connected via switched Gigabit Ethernet using Marvell PCI-E adapter cards.

Communication between the nodes is implemented using the Message Passing Interface (MPI) API. MPI offers synchronous message-based communication primitives and an easy interface to start the application on all nodes and to initialize the communication paths. More specifically, the implementation uses MPICH2 [mpi] which is an implementation of the MPI 2.0 standard from the University of Chicago. MPICH2 provides an Ethernet-based back end for the communication with remote nodes and a fast shared-memory-based back end for local data exchange.

The rest of this section explains the implementation techniques used to parallelize and streamline Wagner's attack to make the best of the available hardware.

7.4.1 Parallelization

Most of the time in the attack is spent on determining the right clamping constants. As described in Section 7.3 this involves computations of several partial trees, e.g., the computation of $L_{3,0}$ from lists $L_{0,0}, \dots, L_{0,7}$ (half tree) or the computation of $L_{2,2}$ from lists $L_{0,8}, \dots, L_{0,11}$ (quarter tree). There are also computations that do not start with lists of level 0; $L_{3,1}$ for example is computed from the (previously computed and stored) lists $L_{2,2}$ and $L_{2,3}$.

Lists of level 0 are generated with the current clamping constants. On every level, each list is sorted and afterward merged with its neighboring list giving the entries for the next level. The sorting and merging is repeated until the final list of the partial tree is computed.

Distributing data over nodes. This algorithm is parallelized by distributing fractions of lists over the nodes in a way that allows each node to perform sort and merge locally on two lists. On each level of the computation, each node contains fractions of two lists. The lists on level j are split between n nodes according to $\lg(n)$ bits of each value. For example when computing the left half-tree, on level 0, node 0

contains all entries of lists 0 and 1 ending with a zero bit (in the bits not controlled by initial clamping), and node 1 contains all entries of lists 0 and 1 ending with a one bit.

Therefore, from the view of one node, on each level the fractions of both lists are loaded from hard disk, the entries are sorted and the two lists are merged. The newly generated list is split into its fractions and these fractions are sent over the network to their associated nodes. There the data is received and stored onto the hard disk.

Presorting into parts. To be able to perform the sort in memory, incoming data is presorted into one of 512 parts according to the 9 least significant bits of the current sort range. This leads to an expected part size for uncompressed entries of 640 MB (0.625 GB) which can be loaded into main memory at once to be sorted further. The benefits of presorting the entries before storing are the following:

- Sorting a whole fraction, that exceeds the size of the memory, can be decomposed into sorting its presorted parts independently;
- two adjacent parts of the two lists on one node (with the same presort-bits) can be merged directly after they are sorted; and
- compressing entries to value-only representation can save 9 more bits.

Sorting and merging. Sorting the presorted parts in memory uses radix sort. Merging two sorted parts is implemented straightforwardly. If blocks of entries in both lists share the same value on the compared b bits, then all possible combinations are generated: specifically, if a b -bit string appears in the compared positions in c_1 entries in the first list and c_2 entries in the second list then all $c_1 c_2$ xors appear in the output list.

7.4.2 Efficient implementation

Cluster computation imposes three main bottlenecks:

- the computational power and memory latency of the CPUs for computation-intensive applications,
- limitations of network throughput and latency for communication-intensive applications, and
- hard-disk throughput and latency for data-intensive applications.

Wagner's algorithm imposes hard load on all of these components: a large amount of data needs to be sorted, merged and distributed over the nodes occupying as much storage as possible. Therefore, demand for optimization is primarily determined by the slowest component in terms of data throughput; latency generally can be hidden by pipelining and data prefetch.

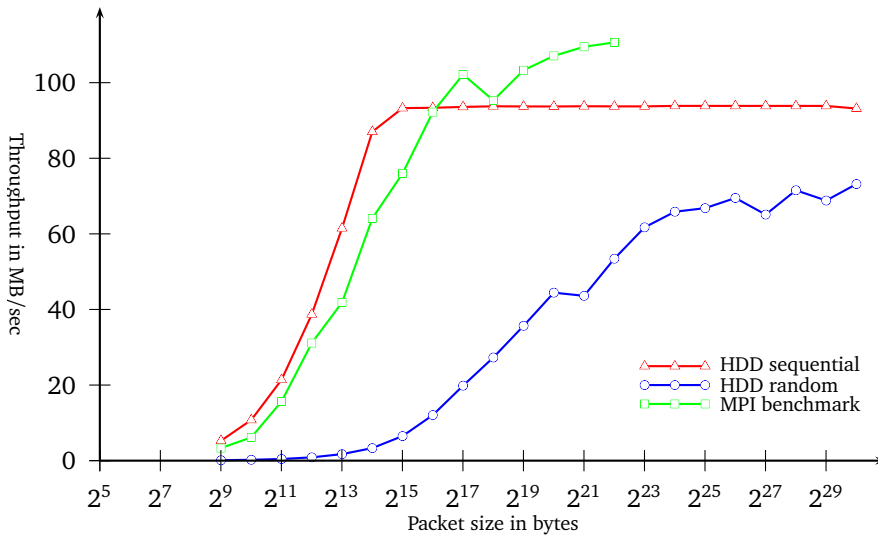


Figure 7.2: Benchmarks measuring hard-disk and network throughput.

Finding bottlenecks. Hard-disk and network throughput of the cluster machines for different packet sizes are shown in Figure 7.2. These numbers were obtained with two self-written micro-benchmarks. Note that the hard-disk benchmark measures hard-disk throughput directly on the device, circumventing the filesystem, to reach peak performance of the hard disk.

The benchmarks show that, for sufficiently large packets, the performance of the system is mainly bottlenecked by hard-disk throughput. Since the throughput of MPI over Gigabit Ethernet is higher than the hard-disk throughput for packet sizes larger than 2^{16} bytes and since the same amount of data has to be sent that needs to be stored, no performance penalty is expected from the network for this size of packets.

Therefore, the highest priority of the implementation is an interface to the hard disk that permits maximum hard-disk throughput. The second goal was to optimize the implementation of sort and merge algorithms up to a level where the hard disks are kept busy at peak throughput.

Persistent data storage. The implementation does not need any caching-, journaling- or even filing-capabilities of conventional filesystems, and thus uses a self-written minimalistic throughput-optimized filesystem. It provides fast and direct access to the hard disk and stores data in fixed-size “files” of 1497600 bytes. This is a multiple of all list-entry sizes occurring in the computations (5 bytes, 9 bytes, 13 bytes, 10 bytes, 20 bytes, and 40 bytes) so that when a part is loaded from disk, it can be treated as a continuous field of entries. Furthermore it is a multiple of 512 bytes, the DMA-transfer block size.

Each cluster node has one large unformatted data partition `/dev/sda1`, which is directly opened by this filesystem using native Linux file I/O. Caching is deactivated by using the open flag `O_DIRECT`: after data has been written, it is not read for a long time and does not benefit from caching. All administrative information is persistently stored as a file in the native Linux filesystem and mapped into the virtual address space of the process. On sequential access, the throughput of this system reaches about 90 MB/s which is roughly the maximum that the hard disk permits.

Tasks and threads. The cluster nodes are driven by quad-core CPUs. To make use of the computational power of all cores the implementation uses multi-threaded parallelization. On the one side, the receiving, presorting and storing, on the other side, the loading, sorting, merging and sending tasks are pipelined. Sending and receiving data is implemented in multiple threads, another thread manages the above-described filesystem. The core of the implementation is given by five threads which process the main computation. There are two threads which have the task of presorting incoming data (one thread for each list). Furthermore, sorting is parallelized with two threads (one thread for each list) and another thread takes care of the merging task.

Memory layout. The benchmarks show that bigger buffers generally lead to higher throughput (see Figure 7.2). However, the sum of all buffer sizes is limited by the size of the available RAM.

- Due to pipelining the list parts need 6 buffers in memory. On level 0 (the worst case) one list containing 2^{37} entries is distributed over 2 nodes and presorted into 512 parts; thus the size of each part should be larger than $(2^{37}/2/512) \cdot 5$ bytes = 640 MB. The actual size of a part must be a multiple of 1497600 bytes, the size of the files the filesystem works with; it is set to $5 \times 9 \times 13 \times 512 \times 5 \times 512 = 766771200$ bytes (731.25 MB) for each part. The six buffers thus require 4.3 GB.
- Double-buffered sending and receiving from and to 8 nodes requires 32 network buffers. Each of these buffers has a size of $2^{20} \cdot 5$ bytes = 5 MB, adding up to total of 160 MB.
- Presorting the entries double-buffered into 512 parts of two lists needs 2048 buffers of size 1497600 bytes each, in total 2925 MB.

Overall the implementation requires about 7.4 GB of RAM leaving enough space for the operating system and additional data as stack and the administrative data for the filesystem.

Efficiency and further optimizations. With the above-described assignment of tasks to threads, the CPU usage is about 60% and reaches a peak of about 80%, the average hard-disk throughput is about 40 MB/s. The hard-disk benchmark (see Figure 7.2) shows that an average throughput between 45 MB/s and 50 MB/s should be feasible for packet sizes of 1.5 MB. With further optimization of the sorting step it would probably be possible to get closer to this maximum hard-disk throughput.

7.5 Results

The implementation described in Sections 7.3 and 7.4 has successfully run and found a collision in the compression function of FSB_{48} . This section presents

- the estimates, before starting the attack, of the amount of time that the attack would need;
- measurements of the amount of time actually consumed by the attack; and
- comments on how different amounts of storage would have changed the attack time.

7.5.1 Cost estimates

Finding appropriate clamping constants. As described before the first major step is to compute a set of clamping values which leads to a collision. In this first step entries are stored by positions on level 0 and 1 and from level 2 on list entries consist of values (see Figure 7.1). Computation of list $L_{3,0}$ takes about 32 hours and list $L_{2,2}$ about 14 hours, summing up to 46 hours. These computations need to be done only once. The time needed to compute list $L_{2,3}$ is about the same as for $L_{2,2}$ (14 hours), list $L_{3,1}$ takes about 4 hours and checking for a collision in lists $L_{3,0}$ and $L_{3,1}$ on level 4 about another 3.5 hours, summing up to about 21.5 hours. The expected value of repetitions of these steps is 16.5. The expected time to finish the first phase of the attack is thus 355 hours.

Computing the matrix positions of the collision. Finally, computing the matrix positions after finding a collision requires recomputation with uncompressed lists. The last two lists $L_{3,0}$ and $L_{3,1}$ do not have to be computed completely but only until the values are found that are known to yield the collision. In the worst case this computation with uncompressed (positions-only) entries takes 33 hours for each half-tree, summing up to 66 hours.

Total expected time. The expected total time to find a collision in FSB_{48} compression function is thus $46 + 355 + 66 = 467$ hours or about 19.5 days.

7.5.2 Cost measurements

The first phase of the attack found clamping constants after only 5 iterations instead of the expected 16.5. In total it took 5 days, 13 hours and 20 minutes.

Recomputation of the positions in $L_{3,0}$ took 1 day, 8 hours and 22 minutes; recomputation of the positions in $L_{3,1}$ took 1 day, 2 hours and 11 minutes. In total the attack took 7 days, 23 hours and 53 minutes.

Recall that the matrix used in the attack is the pseudo-random matrix defined in Section 7.2. The following matrix positions were found to yield a collision in the compression function: 734, 15006, 20748, 25431, 33115, 46670, 50235, 51099, 70220, 76606, 89523, 90851, 99649, 113400, 118568, 126202, 144768, 146047,

153819, 163606, 168187, 173996, 185420, 191473, 198284, 207458, 214106, 223080, 241047, 245456, 247218, 261928, 264386, 273345, 285069, 294658, 304245, 305792, 318044, 327120, 331742, 342519, 344652, 356623, 364676, 368702, 376923, 390678.

7.5.3 Time-storage tradeoffs

As described in Section 7.3, the main restriction on the attack strategy was the total amount of background storage. With 10496 GB of storage at hand the attack could use lists of size 2^{38} , again using the compression techniques described in Section 7.3. This would give exactly one expected collision in the last merge step and thus reduce the expected number of required runs to find the right clamping constants from 16.5 to 1.58. With a total storage of 20 TB a straightforward Wagner attack without compression becomes feasible which eliminates the need to recompute two half trees at the end.

Increasing the size of the background storage even further would eventually allow storing list entry values alongside the positions and thus eliminate the need for dynamic recomputation. However, the performance of the attack is bottlenecked by hard-disk throughput rather than CPU time so this measure would not improve performance.

On clusters with even less background storage the expected number of runs for finding appropriate clamping constants (asymptotically) increases by a factor of 16 with each halving of the storage size. For example a cluster with 2688 GB of storage can only handle lists of size 2^{36} . The attack would then require (expected) 256.5 computations to find appropriate clamping constants.

Of course the time required for one half-tree computation depends on the amount of data. As long as the performance is mainly bottlenecked by hard-disk (or network) throughput the running time is linearly dependent on the amount of data, i.e., a Wagner computation involving 2 half-tree computations with lists of size 2^{38} is about 4.5 times as fast as a Wagner computation involving 18 half-tree computations with lists of size 2^{37} .

7.6 Scalability Analysis

The attack described in this chapter including the variants discussed in Section 7.5 are much more expensive in terms of time and especially memory than a brute-force attack against the full 48-bit hash function FSB_{48} . For example, even poorly-optimized software based on Floyd's cycle-finding algorithm finds a collision in the hash function FSB_{48} in less than 1 minute and 20 seconds on just one core of one cluster node with negligible storage requirements. The collision this software finds is in the two strings `d8 10 d0 d1 f9 a5` and `6e 80 31 0b fd c2`, both producing the hash value `3b 5f 86 1e 22 75`.

Also for the full versions of FSB the designers overestimated the power of Wagner's attack. The resulting poor performance of the FSB hash function in software

Table 7.1: Parameters of the FSB variants and estimates for the cost of generalized birthday attacks against the compression function. For Pollard’s variant the number of lists is marked with a *. Storage is measured in bytes.

	n	w	r	Lists	list size	Bits/entry	Total storage	Time
FSB ₄₈	3×2^{17}	24	192	16	2^{38}	190	$5 \cdot 2^{42}$	$5 \cdot 2^{42}$
FSB ₁₆₀	7×2^{18}	112	640	16	2^{127}	632	$17 \cdot 2^{131}$	$17 \cdot 2^{131}$
				16*	2^{60}	630	$9 \cdot 2^{64}$	$9 \cdot 2^{224}$
FSB ₂₂₄	2^{21}	128	896	16	2^{177}	884	$24 \cdot 2^{181}$	$24 \cdot 2^{181}$
				16*	2^{60}	858	$13 \cdot 2^{64}$	$13 \cdot 2^{343}$
FSB ₂₅₆	23×2^{16}	184	1024	16	2^{202}	1010	$27 \cdot 2^{206}$	$27 \cdot 2^{206}$
				16*	2^{60}	972	$14 \cdot 2^{64}$	$14 \cdot 2^{386}$
				32*	2^{56}	1024	$18 \cdot 2^{60}$	$18 \cdot 2^{405}$
FSB ₃₈₄	23×2^{16}	184	1472	16	2^{291}	1453	$39 \cdot 2^{295}$	$39 \cdot 2^{295}$
				32*	2^{60}	1467	$9 \cdot 2^{65}$	$18 \cdot 2^{618.5}$
FSB ₅₁₂	31×2^{16}	248	1984	16	2^{393}	1962	$53 \cdot 2^{397}$	$53 \cdot 2^{397}$
				32*	2^{60}	1956	$12 \cdot 2^{65}$	$24 \cdot 2^{863}$

(and also hardware) is probably one of the reasons why FSB was not chosen for the second round in the SHA-3 competition.

This section gives estimates of the power of Wagner’s attack against the larger versions of FSB. Table 7.1 gives the parameters of all FSB hash functions.

A straightforward Wagner attack against FSB₁₆₀ uses 16 lists of size 2^{127} containing elements with 632 bits. The entries of these lists are generated as xors of 10 columns from 5 blocks, yielding 2^{135} possibilities to generate the entries. Precomputation includes clamping of 8 bits. Each entry then requires 135 bits of storage so each list occupies more than 2^{131} bytes. For comparison, the largest currently available storage systems offer a few petabytes (2^{50} bytes) of storage.

To limit the amount of memory one can instead generate, e.g., 32 lists of size 2^{60} , where each list entry is the xor of 5 columns from 2.5 blocks, with 7 bits clamped during precomputation. Each list entry then requires 67 bits of storage.

Clamping 60 bits in each step leaves 273 bits uncontrolled so the Pollard variant of Wagner’s algorithm (see Subsection 7.1.2) becomes more efficient than the plain attack. This attack generates 16 lists of size 2^{60} , containing entries which are the xor of 5 columns from 5 distinct blocks each. This gives us the possibility to clamp 10 bits through precomputation, leaving $B = 630$ bits for each entry on level 0.

The time required by this attack is approximately 2^{224} (see (7.3)). This is substantially faster than a brute-force collision attack on the compression function, but is clearly much slower than a brute-force collision attack on the hash function, and even slower than a brute-force *preimage* attack on the hash function.

Similar statements hold for the other full-size versions of FSB. Table 7.1 gives rough estimates for the time complexity of Wagner's attack without storage restriction and with storage restricted to a few hundred exabytes (2^{60} entries per list). These estimates only consider the number and size of lists being a power of 2 and the number of bits clamped in each level being the same. The estimates ignore the time complexity of precomputation. Time is computed according to (7.2) and (7.3) with the size of level-0 entries (in bytes) as a constant factor.

Although fine-tuning the attacks might give small speedups compared to the estimates, it is clear that the compression function of FSB is oversized, assuming that Wagner's algorithm in a somewhat memory-restricted environment is the most efficient attack strategy.

Bibliography

- [ACD⁺06] Roberto Avanzi, Henri Cohen, Christophe Doche, Gerhard Frey, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006. 31
- [AFG⁺09] Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, and Nicolas Sendrier. SHA-3 Proposal: FSB, 2009. <http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=fsb>. 20, 129, 133, 134
- [AFS03] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A fast provably secure cryptographic hash function. Cryptology ePrint Archive, Report 2003/230, 2003. <http://eprint.iacr.org/2003/230/>. 129, 133
- [AFS05] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A family of fast syndrome based cryptographic hash functions. In Serge Vaude- nay, editor, *Progress in Cryptology – Mycrypt 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 64–83. Springer-Verlag Berlin Heidelberg, 2005. <http://lasecwww.epfl.ch/pub/lasec/doc/AFS05.pdf>. 131, 133, 134
- [ALH10] Diego F. Aranha, Julio López, and Darrel Hankerson. High-Speed Parallel Software Implementation of the η_T Pairing. In Josef Pieprzyk, editor, *Cryptographers’ Track at RSA Conference (CT-RSA 2010)*, volume 5985 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag Berlin Heidelberg, 2010. 102
- [ALNR09] Christophe Arène, Tanja Lange, Michael Naehrig, and Christophe Ritzenthaler. Faster computation of the Tate pairing. Cryptology ePrint Archive, Report 2009/155, to appear in the *Journal of Number Theory*, 2009. <http://eprint.iacr.org/2009/155/>. 94
- [ASK06] Onur Aciçmez, Werner Schindler, and Çetin K. Koç. Cache based remote timing attack on the AES. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer-Verlag Berlin Heidelberg, 2006. <http://www.cs.ucsb.edu/~koc/docs/c38.pdf>. 46

- [Bar10] Paulo S. L. M. Barreto. A survey on cryptological pairing algorithms. *Journal of Cryptology*, 7, 2010. http://www.anagram.com/~jcrap/Volume_7/Pairings.pdf. 91
- [BB50] John Bardeen and Walter Houser Brattain. Three-electrode circuit element utilizing semiconductive materials. United States Patent 2,524,035, 1950. Application filed June 17, 1948, <http://www.freepatentsonline.com/2524035.html>. 17
- [BBB⁺07] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management – part 1: General (revised), 2007. Published as NIST Special Publication 800-57, http://csrc.nist.gov/groups/ST/toolkit/documents/SP800-57Part1_3-8-07.pdf. 18, 91
- [BBB⁺09a] Daniel V. Bailey, Brian Baldwin, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Gauthier van Damme, Giacomo de Meulenaer, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Christof Paar, Francesco Regazzoni, Peter Schwabe, and Leif Uhsadel. The Certicom challenges ECC2-X. In *Workshop Record of SHARCS 2009: Special-purpose Hardware for Attacking Cryptographic Systems*, pages 51–82, 2009. <http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf>. 110, 111
- [BBB⁺09b] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier Van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gürkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. *Cryptology ePrint Archive*, Report 2009/541, 2009. <http://eprint.iacr.org/2009/541/>. 110, 113, 127, 128
- [BCC⁺09] Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer-Verlag Berlin Heidelberg, 2009. Document ID: 6904068c52463d70486c9c68ba045839, <http://eprint.iacr.org/2008/480/>. 118
- [BCC⁺10] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology – INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2010. Document ID: 1957e89d79c5a898b6ef308dc10b0446, <http://cryptojedi.org/users/peter/#gpuev11>. 20, 110

- [BCOP04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer-Verlag Berlin Heidelberg, 2004. <http://crypto.stanford.edu/~dabo/papers/encsearch.pdf>. 89
- [Ber] Daniel J. Bernstein. qhasm: tools to help write high-speed software. <http://cr.yp.to/qhasm.html> (accessed 2010-09-07). 38
- [Ber04a] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. Document ID: cd9faae9bd5308c440df50fc26a517b4, <http://cr.yp.to/papers.html#cachetiming>. 37, 46
- [Ber04b] Daniel J. Bernstein. Floating-point arithmetic and message authentication, 2004. Document ID: dabadd3095644704c5cbe9690ea3738e, <http://cr.yp.to/papers.html#hash127>. 97, 101
- [Ber06] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag Berlin Heidelberg, 2006. Document ID: 4230efdfa673480fc079449d90f322c0, <http://cr.yp.to/papers.html#curve25519>. 77, 78, 79, 95, 97
- [Ber07] Daniel J. Bernstein. Better price-performance ratios for generalized birthday attacks. In *Workshop Record of SHARCS'07: Special-purpose Hardware for Attacking Cryptographic Systems*, 2007. Document ID: 7cf298bebf853705133a84bea84d4a07, <http://cr.yp.to/papers.html#genbday>. 131
- [Ber09a] Daniel J. Bernstein. Batch binary Edwards. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer-Verlag Berlin Heidelberg, 2009. Document ID: 4d7766189e82c1381774dc840d05267b, <http://cr.yp.to/papers.html#bbe>. 112, 115, 120
- [Ber09b] Daniel J. Bernstein. Minimum number of bit operations for multiplication, 2009. <http://binary.cr.yp.to/m.html> (accessed 2010-09-07). 115, 116, 122
- [BF01] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag Berlin Heidelberg, 2001. <http://www.iacr.org/archive/crypto2001/21390212.pdf>. 89

- [BF03] Dan Boneh and Matthew Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003. <http://crypto.stanford.edu/~dabo/pubs/papers/bfibe.pdf>. 89
- [BGM⁺10] Jean-Luc Beuchat, Jorge E. González Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. *Cryptology ePrint Archive*, Report 2010/354, 2010. <http://eprint.iacr.org/2010/354/>. 10, 15, 89, 90, 106, 107
- [BGOS07] Paulo S. L. M. Barreto, Stephen D. Galbraith, Colm Ó hÉigeartaigh, and Michael Scott. Efficient pairing computation on supersingular abelian varieties. *Designs, Codes and Cryptography*, 42(3):239–271, 2007. <http://eprint.iacr.org/2004/375/>. 91
- [BGT98] Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms – SODA’98*, pages 574–583. ACM/SIAM, 1998. <http://www.ii.uib.no/~telle/bib/BGT.pdf>. 38
- [Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer-Verlag Berlin Heidelberg, 1997. <http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1997/CS/CS0891.pdf>. 57
- [BK09] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag Berlin Heidelberg, 2009. <http://eprint.iacr.org/2009/317/>. 45
- [BKM09] Joppe W. Bos, Marcelo E. Kaihara, and Peter L. Montgomery. Pollard rho on the PlayStation 3. In *Workshop Record of SHARCS 2009: Special-purpose Hardware for Attacking Cryptographic Systems*, pages 35–50, 2009. <http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf>. 86, 110
- [BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić. Distinguisher and related-key attack on the full AES-256. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 231–249. Springer-Verlag Berlin Heidelberg, 2009. Extended version at <http://eprint.iacr.org/2009/241/>. 45

- [BKNS10] Joppe Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. ECC2K-130 on Cell CPUs. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology – AFRICACRYPT 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag Berlin Heidelberg, 2010. Document ID: bad46a78a56fdc3a44fcf725175fd253, <http://cryptojedi.org/users/peter/#cbev11>. 20, 109, 110, 114, 115
- [BLa] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. <http://bench.cr.yp.to> (accessed 2010-09-07). 63, 65, 72, 86
- [BLb] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. <http://www.hyperelliptic.org/EFD/> (accessed 2010-09-07). 112
- [BL10] Daniel J. Bernstein and Tanja Lange. Type-II optimal polynomial bases. In M. Anwar Hasan and Tor Helleseeth, editors, *Arithmetic of Finite Fields*, volume 6087 of *Lecture Notes in Computer Science*, pages 41–61. Springer-Verlag Berlin Heidelberg, 2010. Document ID: 90995f3542ee40458366015df5f2b9de, <http://cr.yp.to/papers.html#opb>. 113, 114
- [BLN⁺09] Daniel J. Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, and Peter Schwabe. FSBDay: Implementing Wagner’s generalized birthday attack against the SHA-3 round-1 candidate FSB. In Bimal Roy and Nicolas Sendrier, editors, *Progress in Cryptology – INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 18–38. Springer-Verlag Berlin Heidelberg, 2009. Document ID: ded1984108ff55330edb8631e7bc410c, <http://cryptojedi.org/users/peter/#fsbday>. 20, 129, 130
- [BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – Asiacrypt 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer-Verlag Berlin Heidelberg, 2001. <http://www.iacr.org/archive/asiacrypt2001/22480516.pdf>. 89
- [BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004. <http://crypto.stanford.edu/~dabo/pubs/papers/weilsigs.ps>. 89
- [BLS10] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the correct use of the negation map in the pollard rho method, 2010. To appear in proceedings of PKC 2011, Document ID: dde51a91feeb8d746756566ac14323d1, <http://cryptojedi.org/users/peter/#negation>. 86

- [BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer-Verlag Berlin Heidelberg, 2006. http://www.jbonneau.com/AES_timing_full.pdf. 46
- [BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag Berlin Heidelberg, 2006. <http://eprint.iacr.org/2005/133/>. 91, 93
- [Bos10] Joppe Bos. High-performance modular multiplication on the Cell processor. In Anwar Hasan and Tor Helleseth, editors, *Arithmetic of Finite Fields*, volume 6087 of *Lecture Notes in Computer Science*, pages 7–24. Springer-Verlag Berlin Heidelberg, 2010. 86
- [BP09] Joan Boyar and René Peralta. New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191, 2009. <http://eprint.iacr.org/2009/191/>. 59
- [BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer-Verlag Berlin Heidelberg, 2010. 59
- [BR01] Paulo S. L. M. Barreto and Vincent Rijmen. The WHIRLPOOL Hashing Function, 2001. <http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html> (accessed 2010-09-07). 133
- [BS08] Daniel J. Bernstein and Peter Schwabe. New AES software speed records. In Dipanwita Roy Chowdhury and Vincent Rijmen, editors, *Progress in Cryptology – INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer-Verlag Berlin Heidelberg, 2008. Document ID: b90c51d2f7eef86b78068511135a231f, <http://cryptojedi.org/users/peter/#aesspeed>. 20, 41
- [BSS05] Ian Blake, Gadiel Seroussi, and Nigel P. Smart. *Advances in Elliptic Curve Cryptography*. Number 317 in London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, UK, 2005. 90
- [BZB⁺05] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, pages 586–591, Washington, DC, USA,

2005. IEEE Computer Society. <http://home.dei.polimi.it/gpalermo/papers/ITCC05.pdf>. 46
- [Can05] David Canright. A very compact S-box for AES. In Berk Sunar and Josyula R. Rao, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer-Verlag Berlin Heidelberg, 2005. 59
- [Can08] Christophe De Cannière. The eSTREAM project: software performance, 2008. <http://www.ecrypt.eu.org/stream/perf/> (accessed 2010-09-07). 41, 63
- [Cer97a] Certicom. Certicom ECC Challenge, 1997. http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf. 20, 109, 111
- [Cer97b] Certicom. ECC Curves List. <http://www.certicom.com/index.php/curves-list>, 1997. 20
- [CIKL05] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. CryptoGraphics: Secret key cryptography using graphics cards. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 334–350. Springer-Verlag Berlin Heidelberg, 2005. http://www.cs.columbia.edu/~angelos/Papers/2004/gc_ctrnsa.pdf. 118
- [CJ04] Jean-Sébastien Coron and Antoine Joux. Cryptanalysis of a provably secure cryptographic hash function. *Cryptology ePrint Archive*, Report 2004/013, 2004. <http://eprint.iacr.org/2004/013/>. 134
- [CK06] Debra L. Cook and Angelos D. Keromytis. *CryptoGraphics: Exploiting Graphics Cards For Security*, volume 20 of *Advances in Information Security*. Springer Science+Business Media LLC, 2006. 118
- [CRDI05] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine architecture and its first implementation, 2005. <http://www-128.ibm.com/developerworks/power/library/pa-cellperf/>. 75
- [CS07] Neil Costigan and Michael Scott. Accelerating SSL using the vector processors in IBM’s Cell Broadband Engine for Sony’s Playstation 3. In *Workshop Record of SPEED 2007: Software Performance Enhancement for Encryption and Decryption*, pages 65–76, 2007. <http://www.hyperelliptic.org/SPEED/record.pdf>. 86
- [CS09] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer-Verlag Berlin Heidelberg,

2009. Document ID: a33572712a64958c0bf522e608f25f0d, <http://cryptojedi.org/users/peter/#celldh>. 20, 76
- [Dai09a] Wei Dai. Crypto++ 5.6.0 benchmarks, 2009. <http://www.cryptopp.com/benchmarks.html> (accessed 2010-10-20). 42, 72
- [Dai09b] Wei Dai. Crypto++ library, 2009. <http://www.cryptopp.com> (accessed 2010-09-07). 61, 65, 66, 68, 69, 70, 71
- [DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. <http://www-ee.stanford.edu/%7Ehellman/publications/24.pdf>. 18, 76
- [DR99] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. <http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf>. 41, 44
- [DSD07] Augusto J. Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over Barreto-Naehrig curves. In Eiji Okamoto Tsuyoshi Takagi, Tatsuaki Okamoto and Takeshi Okamoto, editors, *Pairing-Based Cryptography – Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 197–207. Springer-Verlag Berlin Heidelberg, 2007. <http://eprint.iacr.org/2007/390/>. 93, 94
- [Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, 2007. <http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>. 41
- [ECR09] ECRYPT II yearly report on algorithms and key sizes (2008-2009). Technical report, ECRYPT II – European Network of Excellence in Cryptology, EU FP7, ICT-2007-216676, 2009. published as deliverable D.SPA.7, <http://www.ecrypt.eu.org/documents/D.SPA.7.pdf>. 18, 91
- [FBB⁺10] Junfeng Fan, Daniel V. Bailey, Lejla Batina, Tim Güneysu, Christof Paar, and Ingrid Verbauwhede. Breaking elliptic curve cryptosystems using reconfigurable hardware. In *2010th International Conference on Field Programmable Logic and Applications (FPL 2010)*. IEEE, 2010. <http://www.cosic.esat.kuleuven.be/publications/article-1460.pdf>. 128
- [Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. <http://ieeexplore.ieee.org/iel5/5/31091/01447203.pdf>. 29

- [Fog10a] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2010. http://www.agner.org/optimize/instruction_tables.pdf (accessed version: 2010-09-25). 28, 55, 98, 106
- [Fog10b] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2010. <http://www.agner.org/optimize/microarchitecture.pdf> (accessed version: 2010-09-25). 28, 55
- [Fog10c] Agner Fog. *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*, 2010. http://agner.org/optimize/optimizing_assembly.pdf (accessed version: 2010-09-25). 28, 55
- [Fre] Freescale. *Altivec*. <http://www.freescale.com/altivec> (accessed 2010-09-07). 30
- [FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010. <http://eprint.iacr.org/2006/372/>. 91
- [GGP09] Philipp Grabher, Johann Großschädl, and Dan Page. On software parallel implementation of cryptographic pairings. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag Berlin Heidelberg, 2009. <http://eprint.iacr.org/2008/205/>. 84, 102
- [Gir08] Damien Giry. Keylength – cryptographic key length recommendation, 2008. <http://www.keylength.com/> (accessed 2010-09-07). 109
- [GLV00] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000. <http://www.ams.org/journals/mcom/2000-69-232/S0025-5718-99-01119-9/S0025-5718-99-01119-9.pdf>. 111
- [gmp] The GNU MP bignum library. <http://gmplib.org/> (accessed 2010-09-07). 97
- [GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 415–432. Springer-Verlag Berlin Heidelberg, 2008. full version: <http://www.brics.dk/~jg/WImoduleFull.pdf>. 89
- [GS10] Robert Granger and Michael Scott. Faster squaring in the cyclotomic subgroup of sixth degree extensions. In Phong Q. Nguyen and David

- Pointcheval, editors, *Public Key Cryptography – PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag Berlin Heidelberg, 2010. <http://eprint.iacr.org/2009/565/>. 94
- [GT07] Pierrick Gaudry and Emmanuel Thomé. The $\text{mp}\mathbb{F}_q$ library and implementing curve-based key exchanges. In *Workshop Record of SPEED 2007: Software Performance Enhancement for Encryption and Decryption*, pages 49–64, 2007. <http://www.hyperelliptic.org/SPEED/record.pdf>. 97
- [Gue08] Shay Gueron. Advanced Encryption Standard (AES) instructions set. Technical report, Intel Corporation, 2008. http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf. 18
- [Har60] Bernard Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, 31:1045–1062, 1960. 111
- [Har00] Robert Harley. Elliptic curve discrete logarithms project. <http://pauillac.inria.fr/~harley/ecdl/> (accessed 2010-09-07), 2000. 110
- [HMS08] Darrel Hankerson, Alfred Menezes, and Michael Scott. Software implementation of pairings. In Marc Joye and Gregory Neven, editors, *Identity-Based Cryptography*. IOS Press, 2008. Draft available online: http://www.math.uwaterloo.ca/~ajmeneze/publications/pairings_software.pdf. 101, 102
- [HMV04] Darrel Hankerson, Alfred Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, 2004. 76, 112, 113
- [HP07] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Elsevier/Morgan Kaufmann Publishers, fourth edition, 2007. 25, 27, 28, 31, 34, 38
- [HSV06] Florian Hess, Nigel P. Smart, and Frederik Vercauteren. The Eta pairing revisited. *IEEE Transactions on Information Theory*, 52(10):4595–4602, 2006. <http://homes.esat.kuleuven.ac.be/~fvercaut/papers/ate.pdf>. 91, 93
- [IBM] IBM. IBM SDK for multicore acceleration (version 3.1). http://www.ibm.com/developerworks/power/cell/downloads.html?S_TACT=105AGX16&S_CMP=LP (accessed 2010-09-07). 115

- [IBM07] IBM DeveloperWorks. SPU assembly language specification (version 1.6), 2007. <http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/EFA2B196893B550787257060006FC9FB>. 79
- [IBM08] IBM DeveloperWorks. Cell Broadband Engine programming handbook (version 1.11), 2008. <https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D>. 79, 115
- [Int03] Intel Corporation. *Intel Pentium 4 Processor and Intel 875P and Intel 850E Chipset Performance Brief*, 2003. www.intel.com/performance/resources/briefs/pentium4.pdf. 27
- [Knu98] Donald E. Knuth. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms*. Addison-Wesley, third edition, 1998. 132
- [KO63] Anatolii Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. Translated from *Doklady Akademii Nauk SSSR*, Vol. 145, No. 2, pp. 293–294, July 1962. 115
- [Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. <http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf>. 75
- [Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO’96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag Berlin Heidelberg, 1996. <http://www.cryptography.com/public/pdf/TimingAttacks.pdf>. 46
- [Kön08] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer-Verlag Berlin Heidelberg, 2008. 58, 71
- [KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag Berlin Heidelberg, 2009. Document ID: cc3a43763e7c5016ddc9cf5d06f8218, <http://cryptojedi.org/users/peter/#aesbs>. 20, 41, 42
- [KSWH00] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2-3):141–158, 2000. <http://www.schneier.com/paper-side-channel2.pdf>. 46

- [KZS⁺09] David Kammler, Diandian Zhang, Peter Schwabe, Hanno Scharwaechter, Markus Langenberg, Dominik Auras, Gerd Ascheid, and Rudolf Mathar. Designing an ASIP for cryptographic pairings over Barreto-Naehrig curves. In *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 254–271. Springer-Verlag Berlin Heidelberg, 2009. Document ID: 7e38974d56cc76a7f572f328ee4a3761, <http://cryptojedi.org/users/peter/#bnchip>. 90
- [Lan] Tanja Lange. Coding and Cryptography Computer Cluster (CCCC). <http://www.win.tue.nl/cccc/> (accessed 2010-12-04). 139
- [Lip06] Helger Lipmaa. AES/Rijndael: speed, 2006. <http://research.cyber.ee/~lipmaa/research/aes/rijndael.html> (accessed 2010-10-15). 65, 66, 67
- [LLP09] Eunjeong Lee, Hyang-Sook Lee, and Cheol-Min Park. Efficient and generalized pairing computation on abelian varieties. *IEEE Transactions on Information Theory*, 55(4):1793–1803, 2009. <http://eprint.iacr.org/2008/040/>. 91
- [Mat06] Mitsuru Matsui. How far can we go on the x64 processors? In Matthew Robshaw, editor, *Fast Software Encryption*, volume 4047 of *Lecture Notes in Computer Science*, pages 341–358. Springer-Verlag Berlin Heidelberg, 2006. <http://www.iacr.org/archive/fse2006/40470344/40470344.pdf>. 66, 71
- [MF05] Mitsuru Matsui and Sayaka Fukuda. How to maximize software performance of symmetric primitives on Pentium III and 4 processors. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 398–412. Springer-Verlag Berlin Heidelberg, 2005. 66
- [Mil86a] Victor S. Miller. Short programs for functions on curves. Unpublished manuscript, 1986. <http://crypto.stanford.edu/miller/miller.pdf>. 91
- [Mil86b] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPTO ’85: Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag Berlin Heidelberg, 1986. 75
- [Mil04] Victor S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17:235–261, 2004. 91
- [MN07] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems –*

- CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer-Verlag Berlin Heidelberg, 2007. 57, 58, 59, 68
- [Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. <http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf>. 106
- [Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. <http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf>. 31, 77, 112
- [Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics magazine*, 1965. <ftp://download.intel.com/research/silicon/moorespaper.pdf>. 17
- [MOVW89] Ron C. Mullin, Ivan M. Onyszchuk, Scott A. Vanstone, and Richard M. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1988/89. 113
- [mpi] MPICH2 : High-performance and widely portable MPI. <http://www.mcs.anl.gov/research/projects/mpich2/> (accessed 2010-09-07). 139
- [MPS10] Andrew Moss, Dan Page, and Nigel P. Smart. Toward acceleration of RSA using 3d graphics hardware. In Steven D. Galbraith, editor, *Cryptography and Coding*, Lecture Notes in Computer Science, pages 364–383. Springer-Verlag Berlin Heidelberg, 2010. <http://www.cs.bris.ac.uk/Publications/Papers/2000772.pdf>. 118
- [MS09] Lorenz Minder and Alistair Sinclair. The extended k -tree algorithm. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 586–595. Society for Industrial and Applied Mathematics, 2009. <http://www.cs.berkeley.edu/~sinclair/ktree.pdf>. 131
- [MV04] David A. McGrew and John Viega. The Galois/Counter Mode of operation (GCM), 2004. <http://www.cryptobarn.com/papers/gcm-spec.pdf>. 20, 41, 44, 47, 61
- [MV05] David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In Anne Canteaut and Kapaleeswaran Viswanathan, editors, *Progress in Cryptology – INDOCRYPT 2004*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer-Verlag Berlin Heidelberg, 2005. full version: <http://eprint.iacr.org/2004/193/>. 72

- [Nae09] Michael Naehrig. *Constructive and Computational Aspects of Cryptographic Pairings*. Ph.D. thesis, Technische Universiteit Eindhoven, 2009. <http://www.cryptojedi.org/users/michael/data/thesis/2009-05-13-diss.pdf>. 90, 91, 92
- [NBB⁺00] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced Encryption Standard (AES). Technical report, National Institute of Standards and Technology, 2000. <http://csrc.nist.gov/archive/aes/round2/r2report.pdf>. 18
- [NBS08] Michael Naehrig, Paulo S. L. M. Barreto, and Peter Schwabe. On compressible pairings and their computation. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 371–388. Springer-Verlag Berlin Heidelberg, 2008. <http://cryptojedi.org/users/peter/#ocpatc>. 90, 102
- [NIS79] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. *Data Encryption Standard (DES)*, 1979. Federal Information Processing Standards Publication 46-3, <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>. 41
- [NIS01] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*, 2001. Federal Information Processing Standards Publication 197, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>. 18, 41
- [NIS07] Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, 2007. http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf. 18
- [NNS10] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Progress in Cryptology – LATINCRYPT 2010*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag Berlin Heidelberg, 2010. Document ID: 4d8d6cd8dc32f9524bb84bbe9c148076, updated version: <http://cryptojedi.org/users/peter/#dclxvi>. 20, 90, 106
- [Nob] The Nobel Prize in Physics 1956. http://nobelprize.org/nobel_prizes/physics/laureates/1956/ (accessed 2010-09-14). 17
- [NS07] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag Berlin Heidelberg, 2007. 46

- [NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, pages 369–369. ACM New York, 2006. <http://palms.ee.princeton.edu/PALMSopen/neve06refined.pdf>. 46
- [OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag Berlin Heidelberg, 2006. <http://eprint.iacr.org/2005/271/>. 46
- [Osv03] Dag Arne Osvik. Fast assembler implementations of the AES, 2003. <http://www.ii.uib.no/~osvik/pres/crypto2003.html> (accessed 2010-10-20). 66
- [Pag02] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002. <http://eprint.iacr.org/2002/169/>. 46
- [Pol78] John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978. <http://www.ams.org/journals/mcom/1978-32-143/S0025-5718-1978-0491431-9/S0025-5718-1978-0491431-9.pdf>. 109, 132
- [SBC⁺09] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Pairing-Based Cryptography – Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*, pages 78–88. Springer-Verlag Berlin Heidelberg, 2009. <http://eprint.iacr.org/2008/490/>. 94
- [Sch28] Arthur Scherbius. Cipherring machine. United States Patent 1,657,411, 1928. Application filed February 6, 1923, Serial No. 617,352, and in Germany February 11, 1922, <http://www.google.com/patents?vid=1657411>. 17
- [Sco07] Michael Scott. Optimal irreducible polynomials for $GF(2^m)$ arithmetic. In *Workshop Record of SPEED 2007: Software Performance Enhancement for Encryption and Decryption*, pages 105–116, 2007. <http://www.hyperelliptic.org/SPEED/record.pdf>. 19
- [Sco10] Michael Scott. Personal communication, March 2010. 102
- [SG08] Robert Szerwinski and Tim Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In Elisabeth Oswald and

- Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer-Verlag Berlin Heidelberg, 2008. http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/publications/conferences/ches2008_szerwinski_etal_asymcryptoongpus.pdf. 118
- [Sha] Shamus Software Ltd. Multiprecision integer and rational arithmetic C/C++ library. <http://www.shamus.ie/> (accessed 2010-09-07). 97, 102
- [Sho51] William Shockley. Circuit element utilizing semiconductive material. United States Patent 2,569,347, 1951. Application filed June 26, 1948, <http://www.freepatentsonline.com/2569347.html>. 17
- [Sho07] Jamshid Shokrollahi. *Efficient implementation of elliptic curve cryptography on FPGAs*. Ph.D. thesis, Rheinische Friedrich-Wilhelms Universität Bonn, 2007. <http://nbn-resolving.de/urn:nbn:de:hbz:5N-09601>. 113
- [Son06] Sony Corporation. *Cell Broadband Engine Architecture, Version 1.01*, 2006. http://cell.scei.co.jp/pdf/CBE_Architecture_v101.pdf. 30
- [TH99] Shreekant (Ticky) Thakkar and Tom Huff. The internet streaming SIMD extensions. *Intel Technology Journal*, Q2, 1999. http://download.intel.com/technology/itj/Q21999/PDF/simd_ext.pdf. 30
- [TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. <http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf>. 46
- [TSS⁺03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzuki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag Berlin Heidelberg, 2003. 46
- [TTMM02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *Proceedings of the International Symposium on Information Theory and Its Applications, ISITA 2002*, pages 803–806, 2002. 46
- [vdL07] Wladimir J. van der Laan. Cubin utilities, 2007. <http://wiki.github.com/laanwj/decuda/>. 126

- [Ver10] Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1), 2010. <http://www.cosic.esat.kuleuven.be/publications/article-1039.pdf>. 91, 92
- [vN45] John von Neumann. First draft of a report on the EDVAC, 1945. 17
- [vN93] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. 17
- [vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999. <http://www.scs.carleton.ca/~paulv/papers/JoC97.pdf>. 110
- [vzGSS07] Joachim von zur Gathen, Amin Shokrollahi, and Jamshid Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68. Springer-Verlag Berlin Heidelberg, 2007. http://cosec.bit.uni-bonn.de/fileadmin/user_upload/publications/pubs/gatsho07.pdf. 113
- [Wag02a] David Wagner. A generalized birthday problem (extended abstract). In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer-Verlag Berlin Heidelberg, 2002. See also newer version [Wag02b]. <http://www.cs.berkeley.edu/~daw/papers/genbday.html>. 20, 129, 130, 163
- [Wag02b] David Wagner. A generalized birthday problem (full version), 2002. See also older version [Wag02a]. <http://www.cs.berkeley.edu/~daw/papers/genbday.html>. 20, 129, 163
- [WPSAM10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *International Symposium on Performance Analysis of Systems and Software - ISPASS*, 2010. <http://www.stuffedcow.net/files/gpuarch-ispas2010.pdf>. 126
- [WZ98] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer-Verlag Berlin Heidelberg, 1998. <http://sites.google.com/site/michaeljameswiener/ECattack.pdf?attredirects=0>. 111

Summary

High-speed Cryptography and Cryptanalysis

Modern digital communication relies heavily on cryptographic protection to ensure data integrity and privacy. In order to deploy state-of-the-art cryptographic primitives and protocols in real-world scenarios, one needs to highly optimize software for both speed and security. This requires careful choices of high-level cryptographic parameters, low-level optimization of software on the assembly level for a given microarchitecture and considerations of the subtle interactions between high-level and low-level optimizations. This thesis considers three examples of cryptographic primitives and describes software implementations of these primitives that set new speed records.

The Advanced Encryption Standard (AES) is one of the most widely used symmetric cryptographic primitives. The traditional implementation approach for AES is based on table lookups. While software based on this approach still achieves best performance for a variety of 32-bit and 64-bit architectures, it is usually vulnerable to cache-timing attacks. Another implementation approach for AES is the bitslicing technique. Not only is software based on this approach inherently protected against cache-timing attacks, on some microarchitectures it even achieves better performance.

Elliptic-curve cryptography is the current state of the art of asymmetric cryptography. For elliptic-curve Diffie-Hellman key exchange, Bernstein proposed the Curve25519 function. Several speed-record-setting implementations of this function have been developed for a variety of architectures. Optimizing Curve25519 software for the Synergistic Processor Units of the Cell Broadband Engine is a particularly interesting challenge because the small integer multipliers of this architecture do not seem to make it the best-suited platform for public-key cryptography.

Another use of elliptic curves in cryptography is in the construction of cryptographic pairings. In order to make pairings fast and secure, very special elliptic curves—so-called pairing-friendly curves—are required. For cryptographic pairings on the 128-bit security level Barreto-Naehrig curves of size about 256 bits are the best choice. Optimizing pairing software is more complex than optimizing, e.g., elliptic-curve Diffie-Hellman key-exchange software. The reason is that pairings involve multiple computation steps and multiple mathematical structures. A choice of parameters considering only some of these steps or structures is likely to incur high performance penalties in the other steps or structures.

Evaluating the security of cryptographic primitives requires cryptanalytic effort. In many ways optimizing cryptanalytic algorithms is similar to optimizing cryptographic primitives: Careful choices of high-level algorithmic parameters such as a Pollard rho iteration function need to be combined with low-level software optimization. A major difference when optimizing cryptanalytic algorithms is the high degree of parallelism that requires additional understanding in optimizing parallel algorithms and in network protocols. This thesis considers two cryptanalytical applications with very different performance bottlenecks and optimization requirements.

Pollard's rho algorithm is the best known algorithm to solve the elliptic-curve discrete-logarithm problem for most prime-order elliptic-curve groups. Large instances of this problem, such as Certicom's challenge ECC2K-130 considered in this thesis, are usually solved using a parallel version of Pollard's rho algorithm, which uses a client-server approach. The efficiency of this approach is mainly determined by the speed of the iteration function, which runs on all clients independently in parallel.

A cryptanalytical algorithm with significantly more complex parallelization requirements is Wagner's tree algorithm. This algorithm involves a huge amount of data for cryptographically relevant inputs; each byte of data needs to be loaded and stored several times. In a parallel environment with distributed storage data cannot be kept local: each byte also needs to be sent various times over the network. The implementation of Wagner's tree algorithm to find a collision in the toy version FSB_{48} of the SHA-3 round-1 candidate FSB on a cluster of 8 computers with a total of 5.5 TB of distributed storage demonstrates techniques to apply this algorithm in storage-restricted environments.

Curriculum Vitae

Peter Schwabe was born on August 20, 1981 in Siegburg, Germany.

In 2000 he graduated from the secondary school Gymnasium am Oelberg in Königswinter. After his community service, he started his studies of computer science at RWTH Aachen University in 2001. He graduated with distinction in computer science in 2006. Peter specialized in cryptography. The title of his Diplomarbeit is “*Efficient Implementation of Elliptic and Hyperelliptic Curves for Cryptographic Applications*”, it was supervised by Prof. Dr. Rudolf Mathar and Michael Naehrig. During his studies he worked as a student research assistant at the Graduate School “*Software for Mobile Communication Systems*” at RWTH Aachen University.

Peter started his Ph.D. studies in October 2006, when he worked as a research and teaching assistant at the Institute for Theoretical Information Technology at RWTH Aachen University. At that time he started research cooperation with Prof. Dr. Tanja Lange.

Starting in January 2008, Peter was a Ph.D. student in the coding theory and cryptology group at Eindhoven University of Technology under the supervision of Prof. Dr. Tanja Lange and Prof. Dr. Daniel J. Bernstein. His work during this period was within the CACE (Computer Aided Cryptography Engineering) project funded by the European Commission. The present dissertation contains the results of his work from 2006 to 2010.

His research interests are in the area of efficient software implementations of cryptographic and cryptanalytical applications.